SANDIA REPORT

SAND2010-2051
Unlimited Release
Printed May 2010

Thyra Coding and Documentation Guidelines

(TCDG)

Version 1.0

Roscoe A. Bartlett
Optimization & Uncertainty Estimation Department

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-2051
Unlimited Release
Printed May 2010

Thyra Coding and Documentation Guidelines

(TCDG)

Version 1.0

Roscoe A. Bartlett
Optimization/Uncertainty Estim

Sandia National Laboratories *, Albuquerque NM 87185 USA,

Abstract

Coding and documentation guidelines help to improve the quality of code and facilitate collaborative development. This document covers C++ coding, code formatting, and Doxygen documentation guidelines that have been established for the Trilinos package Thyra. Many of these guidelines are followed in other Trilinos packages as well. While some of the guidelines outlined in this document are more specifically targeted to Thyra, most of the guidelines are more general than Thyra or even Trilinos.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Contents

Appendix

1 Introduction

This document deals with C++ coding guidelines starting from the foundation of the guidelines in the book "C++ Coding Standards" by Sutter and Alexandrescu [10] (the 101 items are outlined in Appendix C) and the Teuchos-based memory management approach described in [1]. The guidelines in this document are specifically designed to address the development of object-oriented numerical C++ libraries and to utilize the tools in the Trilinos package Teuchos. While the main purpose of this document is to define guidelines for Thyra software (for both interfaces and support software), it is also general enough to be applied to many other projects that, for instance, might interact with Thyra.

The book "C++ Coding Standards" [10] covers many topics that are more general than C++ and can be considered to be general design topics. As a result, the book [10] provides a fairly comprehensive foundation for creating well designed, high-quality C++ software. The goal of this document is not to restate what is in [10] but instead to fill in some gaps intentionally left by the authors and to provide amendments to specific items in [10] and tailor them for numerical libraries. The zeroth item (first item, zero based) "Don't sweat the small stuff" intentionally avoids specific recommendations on issues such as the conventions for naming identifiers and the formatting of code since these are arbitrary. While issues related to coding style are less important than other issues, there are arguments for adopting a more consistent code formatting style and some of these arguments are outlined in Appendix E. Therefore, one of the purposes of this document is to suggest reasonable and minimal guidelines for naming conventions and code formatting that provide for enough code uniformity to facilitate collaborative code development, code reviews, and maintenance in Agile software development processes [3].

More important than code formatting, a consistent set of naming conventions for C++ classes, functions, variables, and other entities also helps to improve collaborative software development and quality. Also, since clients of the software must interact with these names, it is even more important that a set of naming conventions be used as consistently as possible in the client interfaces.

Finally, more important general C++ coding guidelines are covered that append and amend those described in [10]. While formatting and naming recommendations do not affect the meaning of C++ code, other coding guidelines do and therefore they will receive more attention and should be considered more seriously. Unlike naming conventions and code formatting, these guidelines are difficult to change after a significant amount of code has been written.

The rest of the main document is organized as follows. An alpha-numeric convention for naming the various guidelines described in this document is given in Section 2. Then, general naming conventions are presented in Section 3 and they help provide a context for later code examples. This is followed in Section 4 with guidelines for naming and organizing source files. Next, important general C++ coding guidelines are described in Section 5 that affect software quality in critical ways. Following this, reasonable and minimal formatting guidelines are covered in Section 6. Finally, guidelines for Doxygen documentation are provided in Section 7.

Several appendices are included that deal with a number of topics. The guidelines presented in this document are summarized in Appendix A. The 101 guidelines from [10] are listed in Appendix C along with specifying which items are amended or invalidated by the guidelines in the current document. A summary of the idioms and conventions surrounding the use of the Teuchos memory management classes are presented in Appendix B. Appendix D contains discussions of items from [10] that are amended or invalidated in the Thyra coding guidelines. Most importantly, a clarification of `using` declarations is given

that is both more rigid in some ways and less rigid in other ways than what is described in [10, Item 59]. Appendix E gives arguments for adopting a consistent code formatting style in a single development team or single project (which is required with current Agile development methods). Lastly, Appendix F gives guidelines for when one developer can legitimately reformat a source file written by another developer when a more consistent code formatting style is not agreed upon.

2 Alpha-numeric item designations

Specific items in this document are to be refereed to using numerated acronyms starting with and the version number (e.g. 1.0). For example, the first naming convention guideline can be refereed to as **TCDG 1.0 NC 1**. In this way, these short precise alpha-numeric designation such as **TCDG 1.0 NC 3** can be used in code reviews as short-hand references to specific guidelines. The version number of the coding standard is important in order to allow changes in future coding guidelines and allow the numbers to change from version to version (e.g. **NC 1** in **TCDG 1.0** might become **NC 3** in **TCDG X.Y**).

In addition, this document is based on [10] and those guidelines will be refereed to using an enumerated acronym such as **CPPCS Item 15** (i.e. “Use const proactively”).

3 Naming conventions (NC)

C++ classes, functions, variables, data members etc. should be named and used in a fairly consistent manner. The following guidelines are consistent with common practice as exemplified in [8] for example and are also largely consistent with the Java naming standard[1].

- **NC 1**: *Capitalize C++ class and struct names as* `SomeClass`: Names for C++ classes and structs should generally be capitalized and separate words should be concatenated and capitalized (i.e. "Camel Case"). For example:

```
class SomeClass {...};
```

- **NC 2**: *Capitalize C++ namespace names as* `SomeNameSpace`: C++ namespaces should follow the same naming convention as C++ classes and namespace names should not contain too many acronyms and should not be too short or too common. For example:

```
namespace MyNameSpace {
...
} // namespace MyNameSpace
```

- **NC 3**: *C++ enum type names should begin with* `E` *as* `ESomeEnum` *and enum values should use all caps and scope context as* `SOME_ENUM_VALUE`: Enumeration type names should follow the same convention as for class and struct names but they should also begin with the capital letter 'E' to signify that this type is an enum. Enumeration values should be all upper-case with underscores between words and should use a common prefix for scoping within the enum type. Also, enum values should use the default value assignment defined by the compiler in general as this aids their use as indexes into zero-based arrays. For example:

```
enum ESolveStatus {
  SOLVE_STATUS_CONVERGED,
  SOLVE_STATUS_UNCONVERGED,
  SOLVE_STATUS_UNKNOWN
};
```

  Justification: Using a capital 'E' for `enum`s allows the definition of other types with the same basic name that contain other data. For example, `ESolveStatus` in an `enum` enumerating the different types of solve status and `SolveStatus` is a C++ struct that contains an `ESolveStatus` member along with some other data. The use of the scoping prefix (i.e. `SOLVE_STATUS_` above) is also recommended in [7, Section 11.4].

- **NC 4**: *C++ object instance identifier names should begin with a lower-case letter as* `someObject`: Formal function arguments and other object identifiers should, in general, start with a lower-case letter and then use capitalization for following words with no underscores between words in general. For example:

[1] `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

```
ClassType1 obj;
ClassType2 objectForMyThing;
ClassType3 objectForYourThing;
```

Exception: Identifiers that have mathematical symbols in them such as `x`, `J`, and `alpha` should use lower case names separated by underscores `_`. For example:

```
Vector curr_x;
Matrix curr_J;
Scalar curr_alpha;
```

Justification: The Java convention `objectIdentifierName` using capitalization with no underscores produces shorter readable identifiers for English names but does not work well for identifiers with math symbols. With math symbols, it is important to maintain the case of the symbol as `x` and `X` may mean something totally different mathematically and it is confusing and/or ambiguous to write either `currx` or `currX`. In these cases, it is far better to use underscores and write `curr_x` as shown above. While in it is considered bad practice to differentiate variable names by case alone (see "Don't differentiate variable names solely by capitalization" in [7, Section 11.7]), this is very common in math and mathematical software should support this.

- **NC 5**: *C++ class data member names should begin with a lower-case letter and end with an underscore as* `someDataMember_`: Names for data members within a class should use the same naming convention as for other object identifier names but should end with an underscore. For example:

```
class SomeClass {
public:
  ...
private:
  int someDataMember_;
};
```

Justification: Using an underscore after a data variable name helps to define the scope of the variable and differentiate that name from a local variable or a member function that may otherwise result in "shadowing" which causes portability problems on some compilers especially when warnings are elevated to errors.

Exception: Public data members in simple C++ structs (i.e. where no invariants need to be maintained) should not contain underscores. For example:

```
struct SolveStatus {
  ESolveStatus solveStatus;
  double achievedTol;
  std::string message;
  ...
};
```

Exception: Identifiers that have mathematical symbols in them such as `x`, `J`, and `alpha` should use lower case names separated by underscores `_`. For example:

```
Vector curr_x_;
Matrix curr_J_;
Scalar curr_alpha_;
```

Justification: See **NC 4** above.

- **NC 6**: *C++ function names should begin with a lower-case letter as* `someFunction(...)`: Names for functions should use the same naming convention as for object identifier. For example:

```
class SomeClass {
public:
  void someMemberFunction(...);
};

void someOtherFunction(...);
```

Exception: Identifiers that have mathematical symbols in them such as `x`, `J`, and `alpha` should use lower case names separated by underscores `_`. For example:

```
class SomeClass {
public:
  const Vector& get_x() const;
  const Matrix& get_J() const;
  Scalar get_alpha() const;
};
```

Justification: See **NC 4** above.

- **NC 7**: *Name C++ pure abstract base classes* `BlobBase`*, default implementation base classes* `BlobDefaultBase`*, and default concrete implementation classes* `DefaultTypeABlob`: In general, the top-level C++ base class for some abstraction should use the post-fix `Base` prepended to the class name (e.g. `VectorBase`) and the base class should contain (almost) no implementations and certainly no object data (see Item 36 in [10]). If a default implementation of some of the aspects of the base class are desired (to make it easier to define concrete subclasses), then they should be put in a derived node subclass with the post-fix `DefaultBase` (e.g. `VectorDefaultBase`). Any default concrete implementation of an abstraction should generally use the prefix `Default` prepended to the beginning of the name along with any other important prefixes (e.g. `DefaultSpmdVector`). For example:

```
// Pure virtual base class
class VectorBase
  : ... // Other base classes
{
public:
  virtual void applyOp(...) const = 0;
  ...
};

// Node base class with some default implementations
class VectorDefaultBase
 : virtual public VectorBase
```

```
{
public:
  void applyOp(...) const; // default implementation
  ...
};

// A general default implementation for SPMD vectors
class DefaultSpmdVector
  : virtual public VectorDefaultBase // use some default implementations
{
public:
  void applyOp(...) const; // Specialized overrides
  ...
private:
  ...
};
```

- **NC 8**: *Prefer to name const and non-const access functions as* `getPart()` *and* `getNonconstPart()`*, respectively*: In general, functions that return objects that are contained within a wrapper object should have the prefix `Nonconst` added to the function that returns the non-const reference (or pointer) to the contained object. For example,

  ```
  class SomeClass {
  public:
    RCP<Part> getNonconstPart();
    RCP<const Part> getPart() const;
    ...
  };
  ```

  Justification: The choice to name the access functions `getNonconstPart()` and `getPart()` as opposed to `getPart()` and `getConstPart()` is somewhat arbitrary. However, using `nonconst` should be preferred in order to make it more explicit that a non-const object reference is being requested. Also, a constant view of a part of an object is always cheaper that returning a non-constant view of the part (see the discussion of the "generalized view" design pattern in [1]) and therefore to be safe and error on the side of efficiency, the non-constant access function should be harder to call than the constant access function.

4 Naming and organization of source files (NOSF)

Since most C++ code is organized around classes, the file structure should also primarily be organized around classes and the nonmember functions that interact with these classes. The primary goal of these file naming guidelines is to create file names that are globally unique and will therefore facilitate `#includes` without need for directory paths in the `#include` statement. The basic idea is that a source file should be named based on what it has, not where it is. The following guidelines help to define how to organize code into source files and how to name those source files. The directory structure of source files is beyond the scope of this document.

- **NOSF 1**: *Use file extension names `*.hpp` (C++ header), `*.cpp` (C++ source), `*.h` (C header), and `*.c` (C source)*: These file names avoid common problems with portability to various Unix and Windows platforms and enable better tools support (like language-specific formatting in Emacs).
- **NOSF 2**: *Include only one major C++ class with supporting code per header and source file with name(s) `NameSpaceA_InnerNamespace_SomeClass.[hpp,cpp]`*: As a general rule of thumb, assign the source code for any major C++ class and supporting code to a single set of header and source files. The file name should be composed out of the namespace names enclosing the classes and other code along with the class name itself. For instance, for the class `NameSpaceA::InnerNamespace::SomeClass`, the header and source files would be named `NameSpaceA_InnerNamespace_SomeClass.[hpp,cpp]`. This convention assures that the file names will be globally unique. In addition, having a single set of files for each class helps to keep a single encapsulation unit of code together which makes searching the encapsulation unit easier.
- **NOSF 3**: *Use internal include guards in all header files*: All header files, without exception, should use include guards [10, Item 24]. For example, the file `NameSpaceA_InnerNamespace_SomeClass.hpp` would have the basic structure:

```
// @HEADER
// ...
// @HEADER

#ifndef NAMESPACEA_INNERNAMESPACE_SOMECLASS_HPP
#define NAMESPACEA_INNERNAMESPACE_SOMECLASS_HPP

#include "SomeFile.hpp"

...

#endif // NAMESPACEA_INNERNAMESPACE_SOMECLASS_HPP
```

  Above, the comment `// NAMESPACEA_INNERNAMESPACE_SOMECLASS_HPP` after the final `#endif` helps to show the preprocessor structure in the file and is helpful in cases where other `#ifdef` or `#if` structures are used.

  This is a very minor amendment to Item 24 in [10].

5 Coding guidelines

Coding guidelines, unlike formatting guidelines, greatly influence the meaning of C++ programs and therefore require a high priority level. The book "C++ Coding Standards" [10] that this document is primarily based on provides many good and important coding guidelines that should be followed and by default all of the items in [10] are assumed in this document. Here, we provide additional coding guidelines and, in some cases, amend items in [10]. Where this document is silent, [10] is to be considered the authoritative source for guidance. Some miscellaneous amendments to the items in [10] are given in Appendix C and D.

5.1 General coding guidelines (GCG)

Below several different general coding guidelines are discussed. These guidelines affect software quality in a major way and are not just a matter of personal preference or style.

Error handling

- **GCG 1**: *Use `TEST_FOR_EXCEPTION(...)`, `TEUCHOS_ASSERT(...)` and related macros for reporting all errors, even developer programming errors*: For developer errors, prefer to throw exceptions derived from `std::logic_error` instead of using the `assert(...)` macro as recommended in [10, Item 68]. A "logic error" would be treated differently from a real run-time error and would therefore come with different assumptions about the state of the object after the exception was thrown. In particular, a "real error" (i.e. not just an internal developer error) should always provide the basic guarantee to leave the object in a valid state [10, Item 71], while code that throws a "logic error" cannot make any such guarantees in general. Therefore, objects that throw exceptions derived from `std::logic_error` should generally be viewed as unusable and should be deleted immediately. To debug exceptions, a break-point can be placed on function `TestForException_break()`[2] which will be called just before an exception is thrown through these macros. In the future, more sophisticated features like automatically attaching a debugger or printing the call stack may be added for some systems. Therefore throwing an exception derived from `std::logic_error` using these macros should be preferred to using the `assert(...)` macro as it gives us more control over what happens when one of these types of programming errors occurs. Also, these exception macros make it much easier to generate better error messages than what you would get from a simple use of the `assert(...)` macro.

Memory management

- **GCG 2**: *Avoid the use of raw C++ pointers in all but the very specialized siturations*: The Teuchos memory management approach described in [1] mentioned below which include all of the standard C++ container classes (when using a checked STL implementation), `Teuchos::Ptr`, `Teuchos::RCP`, `Teuchos::Array`, `Teuchos::ArrayRCP`, and `Teuchos::ArrayView` allow all code to be written without any explicit raw C++ pointers. In debug mode, these classes provide full run-time checking that result in exceptions being thrown and excellent error messages (i.e. instead of

[2] In `gdb`, a break-point would be set as `b TestForException_break()`.

segfaults). When a check C++ standard library is used (e.g. when _GXXLIB_DEBUG is defined with g++), then all of the standard C++ library classes are checked as well.

- **GCG 3**: *Use* `std::string` *instead of* `char*` *or* `const char*`: While `std::string` is not debug checked in a typical implementation, indexing and other unchecked operations with `std::string` objects is much less common in numerical code and therefore is less likely to result in memory-usage errors inside of numerical code. However, when a checked C++ library implementation is used (e.g. when _GXXLIB_DEBUG is defined with g++), then `std::string` is very safe.

- **GCG 4**: *Use* `Teuchos::Ptr` *as function arguments and return types in the place of raw C++ pointers to single objects for non-persisting and semi-persisting associations*: (see Tables 3 and 4): The class `Teuchos::Ptr` simply takes the place of a raw pointer to a single object but is always default initialized to NULL. In debug mode, it throws exceptions when trying to dereference a null pointer. Using this class helps to eliminate the need for checking for NULL to avoid undefined behavior when one dereferences a NULL pointer.

- **GCG 5**: *Use* `Teuchos::RCP` *for memory management of single dynamically allocated objects and for handling persisting associations*: (see Tables 3 and 4): Replace all references to the class `boost::shared_ptr` in all items in [10] with `Teuchos::RCP`.

- **GCG 6**: *Use non-member constructors for all reference-type classes to force dynamic allocation returning strong owning* `Teuchos::RCP` *objects*: Using non-member constructors gives greater flexibility in how a class object is initialized, simplifies the maintenance of the class, and makes the debug-mode node tracing checking bullet-proof [1].

  Non-member constructors take the form:

```
class SomeClass {
public:
  // No public constructors!
  ..
};

// Non-member constructor
RCP<SomeClass> someClass(...);
```

- **GCG 7**: *Specify "generalized view" semantics for all views of abstract objects*: Using "generalized view" semantics leads to the greatest implementation freedom and the best performance in all cases; abet with more strict usage patterns (see the "generalized view" design pattern in [1]).

  If `SomeBaseClass` provides a view if itself as `Part` objects, then applying the generalized view design pattern results in the interface functions:

```
class SomeBaseClass {
public:
  virtual RCP<Part> getNonconstPart() = 0;
  virtual RCP<const Part> getPart() const = 0;
  ...
};
```

The "generalized view" design pattern along with a concrete example from Thyra is described in great detail in [1].

Note that views of concrete classes do not have to use "generalized view" semantics and can instead use "direct view" semantics where appropriate. See all the details about the "non-member constructor" idiom in [1].

- **GCG 8**: *Use `Teuchos::ArrayView` as function arguments and return types in the place of pointers into raw arrays or other container classes for non-persisting and semi-persisting associations and where the array does not need to be resized*: (see Tables 3 and 4): This class allows all of the useful capabilities of a `std::vector` which do not include adding or removing entries. In debug mode, all of the access functions (including iterators) are fully checked. In optimized mode, unchecked raw pointers are used and the only overhead is a size argument (which is usually passed with raw pointers anyway).

- **GCG 9**: *Use `Teuchos::Array` in place of `std::vector` as a contiguous general purpose data container*: (see Tables 3 and 4): The primary reason to use `Teuchos::Array` instead of `std::vector` is that `Teuchos::Array` is part of the Teuchos system of memory management types and results in stronger run-time checking. While `Teuchos::Array` gets all of its real functionality from `std::vector`, prefer to use `Teuchos::Array` as it provides more capabilities and portable debug checking. For instance `Teuchos::Array::operator[]` is range checked in debug mode regardless whether there is an underling checked STL implementation or not (see [10, Item 83]). In debug mode, the iterator is also run-time checked. In addition, `Teuchos::Array` will automatically convert into an `Teuchos::ArrayView` object safely when used in function calls and in debug mode, will catch dangling references [1].

- **GCG 10**: *Use `Teuchos::ArrayRCP` for memory management of dynamically allocated objects stored in contiguous arrays of data and for persisting associations involving contiguous arrays*: (see Tables 3 and 4): Note that `Teuchos::ArrayRCP` does not take the place of a contiguous container class such as `Teuchos::Array`. A `Teuchos::ArrayRCP` object cannot change the size of the array, it can only provide for reference-counted sharing of an array of data of fixed size and provide sub-views of contiguous parts of the managed array. All access to data (both through `Teuchos::ArrayRCP::operator[]` and iterators) is run-time checked in a debug build.

- **GCG 11**: *Always return `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` smart pointer objects by value, never by reference*: (see Tables 5 and 6): Returning smart pointer objects by value is critical for properly setting up the machinery for persisting and semi-persisting associations and to fully enabled debug-mode checking [1].

- **GCG 12**: *Only return a raw C++ reference from a function for non-persisting associaitons and use the reference and discard it in the same same statement*: (see Tables 3 and 4): Raw C++ references cannot be used to detect dangling references in a debug-mode build and therefore should only be used for non-persisting associations [1].

- **GCG 13**: *Return only `Ptr` and `ArrayView` objects by value to establish semi-persisting associations; never use a raw C++ reference for a semi-persisting association*: (see Tables 3, 4, 5, and 6): Objects of type `Ptr` and `ArrayView` are light-weight and efficient in a non-debug mode build but are fully checked in a debug-mode build and therefore lead to safe efficient code [1].

- **GCG 14**: *When raw C++ pointers must be exposed (i.e., due to interfacing with non-compliant code), minimize the amount of code exposed to the raw pointer*: When raw C++ pointers must be

exposed to communicate with other code that uses raw C++ pointers, encapsulate the raw C++ pointer as fast as possible and then only give up a raw pointer at the last possible moment. For example,

```
SomeForeignClass* get_raw_foreign_obj_ptr();
do_some_foreign_stuff(SomeForeignClass* foreign_obj_ptr);

void foo()
{
  // Get the raw pointer into a proper encapsulated class object right away!
  Ptr<SomeForeignClass> foreignObj(get_raw_foreign_obj_ptr());

  // Lots of code ...

  // Only expose the raw pointer directly in the foreign function call!
  do_some_foreign_stuff(&*foreignObj);
}
```

Object Control

- **GCG 15**: *Accept user options at runtime through a* `Teuchos::ParameterList` *object by deriving from the* `Teuchos::ParameterListAcceptor` *interface*: The `Teuchos::ParameterList` class provides many useful features that make it easy to accept user options in a flexible and fully validated way (see Teuchos documentation for more details). The `Teuchos::ParameterListAcceptor` interface defines a consistent flexible protocol for setting and managing a parameter list.
- **GCG 16**: *Fully validate all parameters and sublists in accepted* `Teuchos::ParameterList` *objects using* `validatePamaters(...)` *and other means*: All user parameters and sub-lists passed in through a `Teuchos::ParameterListAcceptor` should be fully validated. The main tool for this is the member function `validateParameters(...)`. Using this function and other other approaches, when a user misspells a parameter or sub-list, uses the wrong type for a parameter, or provides an invalid parameter value, they will get an exception thrown with a helpful error message. Also, objects are only responsible for validating their own parameters and sub-lists, and not those of other objects that they hold sub-lists for.

Object Introspection

- **GCG 17**: *Always send output to some general* `std::ostream` *object; Never send output directly to* `std::cout` *or* `std::cerr`*; Never print output with* `print(...)` *or* `printf(...)`: Sending output directly to `std::cout` or `std::cerr` destroys the flexibility of numerical software and does not perform well in SPMD programs. Instead, produce output using one of the following approaches.
- *Prefer to print output through a* `Teuchos::FancyOStream` *object instead of through a bare* `std::ostream` *object to more easily produce indented formatted output*: A `Teuchos::FancyOStream` class object can wrap any `std::ostream` object and helps to produce structured indented output and to create more readable output in an SPMD program (even when every processor produces output).

- *Derive from* `Teuchos::Describable` *and implement the functions* `description()` *and* `describe()` *to allow clients to print the current state of an object*: The `Teuchos::Describable` interface is the appropriate way to allow clients to print the current state of an object in a flexible way. The verbosity of the output is controlled by an input enum parameter.

- *Derive from* `Teuchos::VerboseObject` *and print to* `*this->getOStream()` *to give information about what an object is doing*: Clients can set the output stream and the verbosity level through a parameter list (see the `Teuchos::ParameterListAcceptor` interface described above) or can set them directly in code. If no output stream is set, then `Teuchos::VerboseObjectBase::getDefaultOStream()` will be used.

- *As a last resort, always prefer printing to* `*Teuchos::VerboseObjectBase::getDefaultOStream()` *instead of* `std::cout` *or* `std::cerr`: The stream provided by `*Teuchos::VerboseObjectBase::getDefaultOStream()` is set up by default to do clean printing in an SPMD program and can also be set up through a `Teuchos::CommandLineProcessor` object to control how output is produced and formatted.

Miscellaneous coding guidelines

- **GCG 18**: *Prefer to explicitly specify template arguments in a template function call to avoid protability problems and enable implicit covnersions of input arguments*: If it is not too inconvenient, then preferring to explicitly define the template arguments in a template function call can massively improve the portability of templated C++ code. For example, in Thyra, every non-member function is templated on the `Scalar` type such as:

  ```
  template<class Scalar>
  sum(const VectorBase<Scalar> &x);
  ```

  When portability is a concern or when implicit conversions in the input arguments are needed, then prefer to call such functions by specifying the template argument(s) as:

  ```
  Scalar mySum = sum<Scalar>(myVec);
  ```

- **GCG 19**: *Use the template function* `Teuchos::as<T_to>(T_from)` *for all conversion of value data types that may result in loss of precision or in an incorrect conversion*: The templated C++ function `Teuchos::as<T_to>(T_from)` and the class specializations that it calls will contain run-time tests, in debug mode, for the results of a conversion to ensure correctness. This includes the conversion of strings into numbers (i.e. replacing `atof()` and `atoi()`) as well as conversions that can result in loss of precision or meaning (such as `double` to `int`, `long int` to `int`, `int` to `char`, `unsigned int` to `int`, etc.). The optimized implementations of these conversion functions are typically unchecked for speed. A version this function which always does run-time checking is also available called `Teuchos::asSafe<T_to>(T_from)` in order to validate user data.

  Justification: Unchecked conversions are the result of many different types of errors and a fully safe program needs to be able to check all such potentially unsafe conversions at run-time. The implicit conversion rules allowed in C which where carried over to C++ can result in very unsafe code.

- **GCG 20**: *Use namespace enclosure for the definition of C++ class members*: The member functions of a class should be defined in the same order as their declarations and should generally be defined within a namespace enclosure. For example, given the declaration of

```
---------------------------------
// SomeNamespace_SomeClass.hpp

namespace SomeNamespace {

class SomeClass {
public:
  void someFunc();
  ...
};

} // namespace SomeNamespace
---------------------------------
```

the safest and tersest ways to define the member functions in the source file is

```
---------------------------------
// SomeNamespace_SomeClass.cpp

namespace SomeNamespace {

void SomeClass::someFunc()
{
  ...
}

} // namespace SomeNamespace
---------------------------------
```

Justification: Using the namespace enclosure instead of a `using namespace SomeNamesapce` directive insures that you can never accidentally provide another definition for some other class member function in another namespace. Explicit namespace qualification is not needed since if one misspells any part of the prototype, then the compiler will issue an error message.

- **GCG 21**: *Use explicit namespace qualification for the definition of all nonmember C++ functions*: For example, for the nonmember function prototype

```
---------------------------------
// SomeNamespace_someFunc.hpp

namespace SomeNamespace {

void someFunc( const int data );

} // namespace SomeNamespace
---------------------------------
```

the safest way to define the nonmember function is

```
---------------------------------
// SomeNamespace_someFunc.cpp
```

```
void Thyra::someFunc( const int data )
{
  ...
}

---------------------------------
```

Justification: Using explicit namespace qualification avoids problems of spelling and other mistakes that can accidentally result in the definition of a new function [9, Section 8.2]. Such a mistake is caught at link time but it can be very hard to figure out the root cause of the problem when this happens.

- **GCG 22**: *For general functions, prefer to list function arguments in the order of input, input/output, output, and finally optional arguments with default values*: For example:

```
void someFunc(
  const T1 &arg1,         // Input
  const Ptr<T2> &arg2,    // Input/Output
  const Ptr<T3> &arg3,    // Output
  const int arg4 = 0      // Optional input argument with defualt value
  );
```

This ordering of arguments is only a general suggestion as a different ordering of arguments may be chosen based on other criteria. See Section 5.2 for a description of the use of the `Ptr` class.

- **GCG 23**: *For non-member object functions, list the object as the first argument passed in as a const reference or non-const reference*: For example:

```
void someModifyingFunc(
  SomeClass &obj,
  const int arg1,
  ...
  );

void someNonModifyingFunc(
  const SomeClass &obj,
  const int arg1,
  ...
  );
```

Note that in the case of `someModifyingFunc(...)`, the output argument is listed first instead of after the input argument(s) which breaks typical convention of having input/output arguments (which all objects that are modified are) come after input arguments. However, this is more consistent with established convention such as in Python and other languages where the `self` argument is always the first explicit (or implicit) argument. Note that this is also a situation where a non-const reference argument makes the most sense.

- **GCG 24**: *Prefer `enum`s to `bool`s as formal function arguments when conversion mistakes are likely*: While the built-in type `bool` is very convenient to use as a formal function argument, it also allows for conversions from every built-in type and every pointer type. While using an enumeration type and its values is more verbose, it is also self documenting and is safer. For example, what does the third argument mean in the following example?

```
apply( A, 2.0, true, x, y );
```

When the `bool` argument is changed to an enum, the function call becomes:

```
apply( A, 2.0, USE_TRANSPOSE, x, y );
```

and the meaning is much more clear. Therefore, when self documentation and compile-time safety are important, prefer to define and use `enum`s over `bool`s as formal function arguments (see [7, Section 12.6]).

- **GCG 25**: *Avoid overloading virtual functions*: Overloaded virtual functions cause sever portability problems with many compilers and result in shadowing warnings that are elevated to errors in may systems [8, Item 33].

- **GCG 26**: *Avoid overloading functions on different smart pointer types (e.g., `RCP`, `Ptr`, etc.)*: Overloading functions on different smart pointer types, such as `RCP` or `Ptr` can create ambiguous function calls that will not happen when using raw C++ pointers or references [1]. Therefore, keep the names of the functions different such as shown below.

  ```
  void nonconstFoo(const RCP<A> &a);
  void foo(const RCP<const A> &a);
  ```

- **GCG 27**: *Include only standard C++ headers `<cX>`, not standard C headers `<X.h>`, and avoid all `using namespace std` directives*: Only include the C++ `<cX>` versions of the standard C `<X.h>` headers. For example, include `<cmath>`, `<cstdlib>`, and `<cassert>` instead of `<math.h>`, `<stdlib.h>`, and `<assert.h>`. Avoid all uses of `using namespace std` directives and instead prefer explicit namespace qualification such as `std::sqrt` or using declarations such as `using std::sqrt` only within function definitions. See [9, Section 16.1.2] for a complete list of the standard C++ versions of the standard C headers.

  Justification: See Appendix D for a clarification of Item 59 in [10] dealing with the issue of `using` declarations and directives.

- **GCG 28**: *Break up templated code into four files `SomeClass_decl.hpp`, `SomeClass_def.hpp`, `SomeClass.hpp`, and `SomeClass.cpp` to support both implicit and explicit instantiation, minimize recompilation, and avoid problems in mutually dependent (i.e. circular) declarations*: Breaking up templated C++ code into the four files `SomeClass[_decl,_def].[hpp,cpp]` (as described below) allows for a portable and bullet-proof solution to handing templated C++ code which allows for a) controlled explicit or implicit template instantiation, b) minimization of first-time compilation, c) minimization of recompilations, and d) handling of any and all types of circular dependency in declarations and definitions (same as are allowed with non-templated C++ code).

  As an example, consider three classes `A`, `B`, and `C` where `A` and `B` refer to each other and where `C` has no chance of being involved in a circular reference involving `A` and `B`. The four files `A[_decl,_def].[hpp,cpp]` for class `A` as well as the file `B_decl.hpp` are shown below (the other files for class `B` are similar):

```
---------------------------------
// A.hpp

#include "A_decl.hpp"
#ifndef HAVE_THYRA_EXPLICIT_INSTANTIATION
#  include "A_def.hpp"
#endif


---------------------------------
// A_decl.hpp

#ifndef A_DECL_HPP
#define A_DECL_HPP

#include "B_decl.hpp"  // Only include decl in case of circular ref
#include "C.hpp"       // No chance of cicular ref

namespace Thyra {

template<class Scalar>
class A {
pubic:
  void doSomething(const B<T> &b) const;
  ...
private:
  RCP<C<T> > c_;
};

} // namespace Thyra

#endif // A_DECL_HPP


---------------------------------
// B_decl.hpp

#ifndef B_DECL_HPP
#define B_DECL_HPP

namespace Thyra {

template<class Scalar> class A; // Forward only due to circular ref!

template<class Scalar>
class B {
pubic:
  void doSomething(const A<T> &a) const;
};

} // namespace Thyra

#endif // B_DECL_HPP
```

```
---------------------------------
// A_def.hpp

#ifndef A_DEF_HPP
#define A_DEF_HPP

#include "B.hpp" // Must include for implicit instant to work!

namespace Thyra {

template<class Scalar>
void A::doSomething(const B<T>& b)
{
   b.doSomething(*this);
}

} // namespace Thyra

#endif // A_DEF_HPP


---------------------------------
// A.cpp

#include "A_decl.hpp"  // Helps test header sufficiency

#ifdef HAVE_THYRA_EXPLICIT_INSTANTIATION
#include "A_def.hpp"
#include "Teuchos_ExplicitInstantiationHelpers.hpp"
namespace Thyra {TEUCHOS_CLASS_TEMPLATE_INSTANT_SCALAR_TYPES(A)}
#endif // HAVE_THYRA_EXPLICIT_INSTANTIATION
```

General client code always includes the `A.hpp` form of the file without regard for whether implicit or explicit instantiation is enabled or not (i.e. whether `HAVE_THYRA_EXPLICIT_INSTANTIATION` is defined or not defined).

The 100% bullet-proof rules for breaking up template code like this are:

- All header-like declarations that would go into an ordinary non-template `*.hpp` header file go into `SomeClass_decl.hpp` including class declarations and inline function definitions.
- All implementation code that would go into an ordinary non-template `*.cpp` source file go into `SomeClass_def.hpp` including class member definitions and non-member function definitions.
- Always include `SomeOtherClasss_decl.hpp` in the `SomeClass_decl.hpp` file if there is any chance that a circular dependency may exist between the two types `SomeOtherClasss` and `SomeClass`. Otherwise, if there is no chance of a circular dependence then the header `SomeOtherClasss.hpp` should be included (instead of the possible `_decl.hpp` form). If the two classes are in different libraries then there is no chance of a circular type dependency (because well designed software does not allow this [6]).
- If `SomeClass_decl.hpp` includes `SomeOtherClass_decl.hpp`, then `SomeClass_def.hpp` must include `SomeOtherClass.hpp`. This is needed in order for implicit instantiation to work correctly.

– The header file `SomeClass.hpp` is designed to be included by general clients and either includes only `SomeClass_decl.hpp` or also includes `SomeClass_def.hpp` depending on if implicit or explicit instantiation is being used. When explicit instantiation is being used the file `SomeClass_def.hpp` is hidden from general clients and changes in it do not require recompilation of client code. The file `SomeClass.hpp` can (and should) be automatically configured by the CMake build system (see examples in thyra/src/CMakeLists.txt).
– All required instantiations must be provided in the file `SomeClass.cpp`. For standard scalar types (e.g. `double`, `float`, `std::complex<double>`, `std::complex<float>`, etc.) the standard macro `TEUCHOS_CLASS_TEMPLATE_INSTANT_SCALAR_TYPES(...)` is provided which is set at configure time to determine the desired/required explicit instantiations. More general instantiations can also be performed by defining a macro in the file `SomeClass_def.hpp` file and then instantiating this macro using the helper macro `TEUCHOS_MACRO_TEMPLATE_INSTANT_SCALAR_TYPES(...)` (See examples from real Thyra source code).

If one follows the above guidelines, one will never have dependency ordering problems with templated code. The partitioning the template code into the four files `SomeClass[_decl,_def].[hpp,cpp]` gives template code all the desirable compilation properties of non-template code. That is, changes to the implementation of `SomeClass` only require the recompilation of the source file `SomeClass.cpp` and not any other source files. Also, the amount of code that a C++ compiler has to see to compile any single `*.cpp` file is much less when explicit instantiation is enabled and this can massively speed up first-time compilation. Overall, explicit instantiation can massively speed up first-time compilation and later recompilations as code is modified.

5.2 Specification of data members and passing and returning objects from functions

The guidelines for specifying local variables and data members, passing objects to and from functions, and returning objects from functions given in [1] are summarized in Tables 1–6. In general, it is assumed that arguments passed through the smart pointer types `Ptr`, `RCP`, `ArrayView`, and `ArrayRCP` are non-null by default. If the argument is allowed to be null, then that must be documented in the Doxygen `\param` field for that argument.

Class Data Members for Value-Type Objects

Data member purpose	Data member declaration
non-shared, single, const object	`const S s_;`
non-shared, single, non-const object	`S s_;`
non-shared array of non-const objects	`Array<S> as_;`
shared array of non-const objects	`RCP<Array<S> > as_;`
non-shared statically sized array of non-const objects	`Tuple<S,N> as_;`
shared statically sized array of non-const objects	`RCP<Tuple<S,N> > as_;`
shared fixed-sized array of const objects	`ArrayRCP<const S> as_;`
shared fixed-sized array of non-const objects	`ArrayRCP<S> as_;`

Table 1. Idioms for class data member declarations for value-type objects.

Class Data Members for Reference-Type Objects

Data member purpose	Data member declaration
non-shared or shared, single, const object	`RCP<const A> a_;`
non-shared or shared, single, non-const object	`RCP<A> a_;`
non-shared array of const objects	`Array<RCP<const A> > aa_;`
non-shared array of non-const objects	`Array<RCP<A> > aa_;`
shared fixed-sized array of const objects	`ArrayRCP<RCP<const A> > aa_;`
"..." (const ptr)	`ArrayRCP<const RCP<const A> > aa_;`
shared fixed-sized array of non-const objects	`ArrayRCP<RCP<const A> > aa_;`
"..." (const ptr)	`ArrayRCP<const RCP<const A> > aa_;`

Table 2. Idioms for class data member declarations for reference-types objects.

Passing IN Non-Persisting Associations to Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object (required)	`S s` **or** `const S s` **or** `const S &s`
single, non-changeable object (optional)	`const Ptr<const S> &s`
single, changeable object (required)	`const Ptr<S> &s` **or** `S &s`
single, changeable object (optional)	`const Ptr<S> &s`
array of non-changeable objects	`const ArrayView<const S> &as`
array of changeable objects	`const ArrayView<S> &as`

Passing IN Persisting Associations to Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const ArrayRCP<const S> &as`
array of changeable objects	`const ArrayRCP<S> &ss`

Passing OUT Persisting Associations for Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const Ptr<ArrayRCP<const S> > &as`
array of changeable objects	`const Ptr<ArrayRCP<S> > &as`

Passing OUT Semi-Persisting Associations for Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const Ptr<ArrayView<const S> > &as`
array of changeable objects	`const Ptr<ArrayView<S> > &as`

Table 3. Idioms for passing value-type objects to C++ functions.

Passing IN Non-Persisting Associations to Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object (required)	`const A &a`
single, non-changeable object (optional)	`const Ptr<const A> &a`
single, changeable object (required)	`const Ptr<A> &a` **or** `A &a`
single, changeable object (optional)	`const Ptr<A> &a`
array of non-changeable objects	`const ArrayView<const Ptr<const A> > &aa`
array of changeable objects	`const ArrayView<const Ptr<A> > &aa`

Passing IN Persisting Associations to Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const RCP<const A> &a`
single, changeable object	`const RCP<A> &a`
array of non-changeable objects	`const ArrayView<const RCP<const A> > &aa`
array of changeable objects	`const ArrayView<const RCP<A> > &aa`

Passing OUT Persisting Associations for Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const Ptr<RCP<const A> > &a`
single, changeable object	`const Ptr<RCP<A> > &a`
array of non-changeable objects	`const ArrayView<RCP<const A> > &aa`
array of changeable objects	`const ArrayView<RCP<A> > &aa`

Passing OUT Semi-Persisting Associations for Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const Ptr<Ptr<const A> > &a`
single, changeable object	`const Ptr<Ptr<A> > &a`
array of non-changeable objects	`const ArrayView<Ptr<const A> > &aa`
array of changeable objects	`const ArrayView<Ptr<A> > &aa`

Table 4. Idioms for passing reference-type objects to C++ functions.

Returning Non-Persisting Associations to Value Objects

Purpose	Return Type Declaration
Single copied object (return by value)	`S`
Single non-changeable object (required)	`const S&`
Single non-changeable object (optional)	`Ptr<const S>`
Single changeable object (required)	`S&`
Single changeable object (optional)	`Ptr<S>`
Array of non-changeable objects	`ArrayView<const S>`
Array of changeable objects	`ArrayView<S>`

Returning Persisting Associations to Value Objects

Purpose	Return Type Declaration
Array of non-changeable objects	`ArrayRCP<const S>`
Array of changeable objects	`ArrayRCP<S>`

Returning Semi-Persisting Associations to Value Objects

Purpose	Return Type Declaration
Array of non-changeable objects	`ArrayView<const S>`
Array of changeable objects	`ArrayView<S>`

Table 5. Idioms for returning value-type objects from C++ functions.

Returning Non-Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single cloned object	`RCP<A>`
Single non-changeable object (required)	`const A&`
Single non-changeable object (optional)	`Ptr<const A>`
Single changeable object (required)	`A&`
Single changeable object (optional)	`Ptr<A>`
Array of non-changeable objects	`ArrayView<const Ptr<const A> >`
Array of changeable objects	`ArrayView<const Ptr<A> >`

Returning Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single non-changeable object	`RCP<const A>`
Single changeable object	`RCP<A>`
Array of non-changeable objects	`ArrayView<const RCP<const A> >`
Array of changeable objects	`ArrayView<const RCP<A> >`

Returning Semi-Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single non-changeable object	`Ptr<const A>`
Single changeable object	`Ptr<A>`
Array of non-changeable objects	`ArrayView<const Ptr<const A> >`
Array of changeable objects	`ArrayView<const Ptr<A> >`

Table 6. Idioms for returning reference-type objects from C++ functions.

6 Formatting of source code

At the minimum, source code should be formatted consistently within a single file or a set of tightly coupled files [10, Item 0]. Ideally, source code should be formatted consistently enough across a code project so as not to cause undue difficulty in shared maintenance and in performing code reviews [7]. Some consistency in formatting helps and to facilitate multiple ownership and shared development of a collection of software, such as in Extreme Programming (XP) [2] (see Appendix E for an outline of the arguments for adopting a consistent code formatting style). By "formatting" we generally refer to the use of white-space in the line-to-line formatting of the program or in the ordering of lines of code such that the meaning of the program to the compiler is unchanged[3] The handling of indentation styles can largely be automated[4] which allows individual developers to work with any style they would like for files that they create but also makes it easy for developers to edit files created by other developers and keep to their styles as well. Appendix F gives some guidelines for how individuals should conduct themselves where more than one code formatting style is in use within a project.

Our main goal in this section is to try to provide reasonable recommendations for those formatting issues that are largely a matter of style and personal preference but at the same time affect the overall readability of the code and promote pair programming and joint ownership of code [2]. The formatting and indentation guidelines presented here are largely consistent with the recommendations in [7, Chapter 31] and try to reduce the amount of "right drift" that can occur with some common formatting and indentation styles.

The indentation guidelines outlined below can be largely automatically supported by Emacs and are used by the custom style "thyra" defined in the Emacs package file cc-thyra-styles.el[5]. Other custom styles can also be added to this file and used as well. Any of these styles can be listed in each source file and therefore anyone using Emacs can automatically use a particular indentation style without having to fight the editor to manually reformat code to abide by a foreign style.

6.1 General formatting source code principles (FSCP)

Some general principles of good formatting, based on the discussion in [7, Section 31.1], are:

- **FSCP 1**: *Formatting should accurately and consistently show the logical structure of the code*: It is somewhat subjective what formatting styles "show the logical structure" of code but McConnell makes some good arguments for some styles over others. However, it is up the group of programmers to decide as a group what style items "show the logical structure".
- **FSCP 2**: *Formatting should improve the readability of the code for most people*: There are specific studies cited in [7, Chapter 31] that provide good evidence to prefer some styles over others.
- **FSCP 3**: *Formatted code should retain its formatting well when modified; especially for those modifications performed by automated tools*: Changing one line of code should not require changes to other lines of code to maintain the formatting style.

[3] While technically changing the name of a class, function or variable changes the meaning of a program, if name changes are done in such a way as to avoid name collisions, then naming conventions also do not affect the meaning of the program and are therefore very much related to other formatting issues such as the treatment of "white-space".

[4] Emacs supports multiple file-specific formatting styles for C++ and tools like Artistic Style [4] can format source files from the command line. A flavor of the `vi` editor may also support indentation styles.

[5] See `Trilinos/packages/thyra/emacs/README` for a description of the "thyra" Emacs style

- **FSCP 4**: *Formatting style should follow the most common idiom unless one of the above principles are violated*: When there is no good technical argument for one formatting style choice over another, then the style choice that is the most common should be used[6]. This is not advocated per-say in [7, Chapter 31] but it is a good idea in general to follow popular idioms when there are several equally good choices and therefore the decision is arbitrary. However, not selecting a single style choice can create artificial complexity in the code from irregularity in formatting.

6.2 Specific guidelines for formatting source code (FSC)

Below, specific recommendations are spelled out that try to conform to common practices but also try to avoid excessive "right drift":

- **FSC 1**: *The formatting style in any single file or group of closely related files should be the same*: Consistent formatting includes the placement of braces, the number spaces to indent etc. *Justification*: This is recommended in [10, Item 0].
- **FSC 2**: *Try to keep all text within the first 80 character columns*: Keeping most of the source code within the first 80 character columns helps to make the code more readable and helps to facilitate side-by-side two-column editing and comparisons of source code. Most of the style and indentation guidelines described below help to avoid code that extends beyond the 80th column too rapidly. *Justification*: "Studies show that up to ten-word text widths are optimal for eye tracking" [10, Item 0]. Also, some developers are still stuck with 80 column wide terminals.
- **FSC 3**: *Indent with spaces and not tabs (two spaces by default)*: The amount of spaces to use per indentation level is up to the individual developer but an indentation of only *two spaces* is recommended (and is set in the 'Emacs 'thyra" indentation style). A study showed that an indentation offset of two-to-four spaces was optimal for code reading comprehension [7, Section 31.2]. Whatever indentation amount is used, it should be consistent in at least each source and header file [10, Item 0] (which can be enforced using a custom Emacs indentation style). Emacs by default will put in a tab when the tab-width is equal to the number of indentation spaces. Emacs can be told to always use spaces instead of tabs by setting:

  ```
  (setq indent-tabs-mode nil)
  ```

  in the indentation style (as is done in the "thyra" style). However, it is easy to support different preferences for the amount of spaces to indent by using a user-defined indentation style for Emacs (sorry `vi` users).

  Justification: "Some teams legitimately choose to ban tabs ... when misused, turn indenting into out-denting and non-denting." [10, Item 0].

- **FSC 4**: *Use two vertical spaces to separate class declarations, function definitions, namespace enclosure bounds, and other such major entries in a file.*

  Justification: Using two black spaces is preferable to long lines with some filler like '-' or '=' or other separators and they clearly separate the entities and are easier to maintain (see [7, Section 31.8]).

[6] The measure of the commonality of a particular style choice can be determined according to a local software development community or the larger developer community.

- **FSC 5**: *Do not indent source code inside of namespace enclosures, instead use commented end braces*: Indenting for namespace enclosures results in unnecessary, and in some cases excessive, indentation. Instead, for example, use:

```
namespace MyNameSpace {


namespace MyInnerNamespace {


class SomeClass {..};


void someFunc(...) {...}


} // namespace MyInnerNamespace


} // namespace MyNameSpace
```

  Above, note that two vertical blank lines are used between each of the major entities (see above item).

  Justification: While indentation within namespaces is helpful in small example code fragments, it provides little help in showing namespace structure in more realistic code. The use of commented end braces is generally sufficient to show namespace structure and will not result in excessively indented code. In addition, typically, each file will only contain code from one (or more nested) namespace and therefore indenting for namespaces provides no useful information. Not indenting for namespace enclosures is also consistent with the "ansi", the "kr", and the "linux" styles as defined by Artistic Style [4].

- **FSC 6**: *C++ class declarations should generally be laid out with* `public` *members coming before* `protected` *members coming before* `private` *members and indented as shown in Figure 1.*

  Justification: This ordering of sections and data members is quite common [7, Section 31.8]. Above, we show private member functions after private data members since private data members are more prominent and more common in the class implementations than are private member functions. Also, private types (where typedefs are most common) must be listed before they are used in the declaration of the private data members. Note that public types used in public member functions must be listed above (or at least forward declared) before the public member functions that use them.

- **FSC 7**: *List short function prototypes on one line and longer prototypes on multiple lines, indenting arguments one unit*: Below, guidelines for formatting short function prototypes and long prototypes are given. These guidelines seek to produce function prototypes that are fairly tight (i.e. not too much white-space explosion), are robust to modifications, and keep code inside of the 80th character column. This indentation style can (and should) also be applied to function definitions and function calls.

  – *List short function prototypes on one line if possible*: For example,

```
ReturnType someFunction( int arg = 0 );
```

```
class SomeClass {

  // Friends

  friend void foo();

  friend class SomeOtherClass;

public:

  // Public types

  typedef int integral_type;

  // Public member functions

  void func1();

protected:

  // Protected member functions

  void func2();

private:

  // Private types

  typedef std::vector<int> int_array_t;

  // Private data members

  int data1_;
  int_array_t array1_;

  // Private member functions

  void func3();

};
```

Figure 1. Example of suggested layout of a C++ class declaration complete with ordering of sections, indentation, and line spacing.

or

```
ReturnType someFunction(int arg=0);
```

or some other style for white-space within '(...)' but the opening '(' should come directly after the function name in all cases.

– *For longer prototypes, indent arguments on continuation lines one unit*: Function prototypes that cannot approximately fit on a single line in the first 80 character columns should have the function arguments listed starting on the second line with one unit of indentation (e.g. two spaces) from the function return type and function name line. For example, several different valid formats for a longer function prototype are:

```
ReturnType someFunction(
  int arg1,
  bool arg2,
  const ArrayView<double> &arg3,
  const std::string &arg4  = ""
  );
```

or

```
ReturnType someFunction(
  int arg1, bool arg2, const ArrayView<double> &arg3,
  const std::string &arg4  = ""
  );
```

or

```
ReturnType someFunction(
  int arg1, bool arg2, const ArrayView<double> &arg3,
  const std::string &arg4 = "" );
```

or

```
ReturnType someFunction( int arg1, bool arg2,
  const ArrayView<double> &arg3, const std::string &arg4 = "" );
```

As shown above, the function arguments can be listed separately on different lines, or in groups on sets of lines. The arguments can begin on the same line as the type + function name line or can start on the next line. The ending parenthesis ')' can appear on the same line as the last line of arguments or can appear alone on the last line. Other formats are possible also and can be appropriate in different situations.

Justification: See [7, Section 31.1].

– *Return types can be listed on same line as the function name unless the line is too long*: A function prototype's return type should appear on the same line as the function name unless it is excessively long and would result in the return type + function name line to extend past the 80th character column. When the return type + function name is too long, then it can be listed on separate lines with no indent, for example, as:

```
Teuchos::RCP<ReturnType>
someVeryLongAndVeryImportantFunction(
  int arg1, bool arg2, const ArrayView<double> &arg3,
  const std::string &arg4  = ""
  );
```

However, listing the function return type on a separate line even in cases of shorter prototypes is also okay.

- **FSC 8**: *Order the definitions of C++ entities the same as the order of the declarations of those entities*: For example, one should order the definitions of a set of functions the same as the ordering of the declarations. Maintaining the ordering of definitions and declarations makes the code more readable and more maintainable. For example, if the function definitions are ordered the same as the declarations, it can be easy to spot that a function definition is missing (i.e. which could be the cause of the link error that you are seeing).

- **FSC 9**: *Use "modified K&R" or "ANSI" style for the placement of braces and indentation of control structures*: Two basic styles of brace placement and indentation in control structures are recommend here. The first general style is a modification of the K&R style[4] where the brace comes immediately after the control statement on the same line shown as:

```
// Modified K&R Style (recommended)
if (someCondition) {
  ...
}
else {
  ...
}
```

Note that the pure K&R style (for example, as defined by Artistic Style [4]) shown as:

```
// Pure K&R Style (*NOT* recommended)
if (someCondition) {
  ...
} else {
  ...
}
```

is not recommended. Even through pure K&R style meets McConnell's strict pictorial definition of "emulation of pure block style" (i.e. the equivalent to pure block format such as in Visual Basic) which he says is good, he actually recommends the above modified K&R style (as do we since we feel it is more readable).

The second general style that is recommended is the "ANSI" style[4] where the opening brace begins flush on the next line from the control statement shown as:

```
// ANSI Style (recommended)
if (someCondition)
{
  ...
```

```
}
else
{
  ...
}
```

Both the modified K&R and the ANSI styles help to avoid right drift. The modified K&R style creates tighter code vertically and seems to be preferred by many communities and authors but variations of the ANSI style are also very common. Note that the ANSI style seems to have a distinct advantage in cases where the control statement is continued over multiple lines. For example, the modified K&R style with line continuations looks like:

```
// Modified K&R Style with line continuations (*NOT* recommended)
if ( someLongCondition &&
  anotherVeryLongCondition &&
  theLongestConditionThatWillFitOnOneLine ) {
  // Statements
  ...
}
```

and it is hard to argue that this shows the logical structure of code. One could argue that the ANSI style which looks like:

```
// ANSI Style with line continuations (recommended)
if ( someLongCondition &&
  anotherVeryLongCondition &&
  theLongestConditionThatWillFitOnOneLine )
{
  // Statements
  ...
}
```

better shows the logical structure of the code in clearly separating the control structure logic from the inner block of code.

Note that while the modified K&R style meets McConnell's blessing of "showing the logical structure of code" where he refers to it as "emulating pure block" format that he cites the ANSI styles as violating this principle [7, Section 31.1]. However, it is somewhat subjective what styles "show the logical structure" and McConnell himself seems to contradict himself at times (see the formatting of if/else statements below).

When choosing between one of these styles, try to be consistent at least within a single file. However, for control statements that extend over a single line, prefer the "ANSI" style.

Below, the application of the modified K&R style and the ANSI styles are shown in the context of several different types of C++ loop and control structures.

- *Formatting if/else if/else statements*: When applied to if statements, the two recommended styles are:

```
// Modified K&R Style (recommended)
if (someCondition) {
  ...
}
else if (someOtherCondition) {
  ...
}
else {
  ...
}
```

and:

```
// ANSI Style (recommended)
if (someCondition)
{
  ...
}
else if (someOtherCondition)
{
  ...
}
else
{
  ...
}
```

– *Formatting switch/case statements*: The two recommended formats for switch/case statements are:

```
// Modified K&R Style (recommended)
switch (someEnumValue) {
  case ENUM_VALUE1:
    ...
    break;
  case ENUM_VALUE2:
    ...
    break;
  default:
    TEST_FOR_EXCEPT("Should never get there!");
}
```

and

```
// ANSI Style (recommended)
switch (someEnumValue)
{
  case ENUM_VALUE1:
    ...
    break;
  case ENUM_VALUE2:
    ...
```

```
    break;
  default:
    TEST_FOR_EXCEPT("Should never get there!");
}
```

As shown above, every switch structure should have a `default` case that throws an exception (see "use the default clause to detect errors" in [7, Section 15.1]).

Also, if needed, the case blocks can be wrapped in braces as:

```
// Modified K&R Style (recommended)
switch (someEnumValue) {
  case ENUM_VALUE1: {
    ...
    break;
  }
  case ENUM_VALUE2: {
    ...
    break;
  }
  default: {
    TEST_FOR_EXCEPT("Should never get there!");
  }
}
```

and

```
// ANSI Style (recommended)
switch (someEnumValue)
{
  case ENUM_VALUE1:
  {
    ...
    break;
  }
  case ENUM_VALUE2:
  {
    ...
    break;
  }
  default:
  {
    TEST_FOR_EXCEPT("Should never get there!");
  }
}
```

– *Formatting for and while loops*: The two recommended styles for formatting for loops are:

```
// Modified K&R Style (recommended)
for ( int i = 0; i < size; ++i ) {
  ...
}
```

and:

```
// ANSI Style (recommended)
for ( int i = 0; i < size; ++i )
{
  ...
}
```

Note that line continuations are often needed for a for loops control structure, especially if long type names or variable names are used. In these cases, the ANSI style is more highly recommended as:

```
// ANSI Style (recommended)
for (
  std::vector<SomeVeryLongClassName>::const_iterator itr = longVarName.begin();
  itr != someLongVariableName.end();
  ++itr )
{
  ...
}
```

Similarly, while loops should be formatted as:

```
// Modified K&R Style (recommended)
while ( someCondition ) {
  ...
}
```

or:

```
// ANSI Style (recommended)
while ( someCondition )
{
  ...
}
```

7 Doxygen documentation guidelines

In this section, a set of reasonable guidelines are stated for writing Doxygen (and plain old) documentation for classes, functions, etc. that makes the specification clear but is not too verbose or hard to maintain. While other types of higher-level documentation are also needed such as design documents and tutorials, guidelines for these other types of higher-level documentation are not covered here.

7.1 General principles for function and class level documentation (DOXP)

- **DOXP 1**: *The level of documentation should vary depending on the prominence and/or the role of the software entity or collection*: Important interfaces or widely disseminated concrete classes or functions require an appropriate level of precise documentation. Concrete implementations that are less widely disseminated can provide less (or none in some cases) Doxygen documentation if the implementation code itself is sufficiently easy to understand. However, major parts of an implementation should have at least some plain old (i.e. non-Doxygen) documentation to describe the basics of what is going on.

- **DOXP 2**: *Important abstract interfaces must be fully specified independent of any single concrete implementation (i.e. preconditions, postconditions, invariants, etc.)*: In the case of important abstract interfaces, the full specification of behavior for the compliant objects (i.e. invariants, preconditions, postconditions) must be clearly stated [10, Item 69]. In some cases, this must be done completely within the Doxygen documentation for the interface. In other cases, standard unit testing code can be used to help specify the behavior of the interface. In fact, compiled and verified unit testing code may be superior to standard Doxygen documentation since it cannot be ignored and cannot become invalid. On the other hand, it may be difficult for readers to wade through unit testing code to find the specification of behavior and therefore both Doxygen documentation and unit testing code should be used to provide the fullest benefit. Also, Doxygen documentation can automatically include bits and pieces of compiled and tested code using the `\dontinclude` and related Doxygen commands.

- **DOXP 3**: *Behavior of "user level" interfaces must be completely specified by the Doxygen documentation and/or other higher-level documentation (i.e. preconditions, postconditions, invariants, etc.)*: This item is an amendment to the above item as a special case for "user" interfaces. A "user" could be someone that simply writes client code to the interface or one that provides implementations of the interface or both. User's should not be expected to study unit testing code to figure out the preconditions and/or postconditions for a function call.

- **DOXP 4**: *Wrong documentation is (almost) worse than no documentation at all*: Documentation must be maintained as code is changed and therefore excessive or unnecessary documentation that is not rigorously maintained degrades the overall quality of code. However, documentation with small errors is generally better than no documentation at all.

- **DOXP 5**: *The same documentation should not be repeated in more than one place if possible*: We should strive for a single source for documentation for an entity and not repeat the same documentation over and over again. This is critical to insure that the documentation can be successfully maintained.

- **DOXP 6**: *The documentation should maintain itself as much as possible and be testable as much as possible*: Any significant fragments of code that are shown in the Doxygen-generated HTML

documentation should come from compiled and tested code. This can be accomplished by using the \dontinclude or related Doxygen command to read in code fragments automatically. In this way, the compiler and our test suite can be used to help verify the code fragments in our Doxygen documentation.

7.2 Specific Doxygen documentation principles (DOX)

Now that some of the general goals for our Doxygen documentation have been presented, more detailed guidelines are given below.

- **DOX 1**: *Write Doxygen documentation directly in header files with documented entities*: Writing Doxygen documentation comments directly attached to the classes, functions and other entities helps make the documentation as tightly tied to the code as possible (see "Keep comments close to the code they describe" in [7, Section 32.5]). This has the unfortunate side-effect of requiring complete recompilations whenever documentation is modified but the overall benefits are usually worth the disadvantages. Note that the Doxygen documentation can be stripped out of Doxygen-generated hyper-linked versions of the code, leaving clean C++ code without the clutter of detailed documentation. Therefore, developers should browse Doxygen-generated source code instead of the source code directly when looking at the code and performing code reviews.

- **DOX 2**: *Use a centralized set of definitions for common arguments whenever possible*: Use clear and consistent naming of arguments in multiple functions (within the same class and across as many classes and functions as makes sense) and provide a centralized definition of these arguments if possible to avoid repeating detailed descriptions in each individual function's documentation. This helps to avoid duplicate documentation that is likely not to be maintained correctly. In the case of classes, this means providing some common definitions in the main "detailed" documentation section for the class. In the case of nonmember functions, this might involve a common Doxygen group or module (i.e. using the \defgroup command) for the set of functions. In the case of collections of nonmember functions, it may be difficult to expect readers to find the common definitions, but links to the common documentation are possible using a variety of approaches.

- **DOX 3**: *Provide typical pre- and postconditions along with the documentation for common arguments whenever possible*: For common arguments that are shared among many functions, define the most common preconditions for them in a central place and avoid listing them on a function-by-function basis unless they change for an individual function. For a C++ class, place descriptions for these common arguments in the main class documentation under a \section named "Common Function Arguments and Pre/Postconditions". Only include preconditions for these arguments in specific function documentation sections if it is different from the most common preconditions.

- **DOX 4**: *Add a \brief description for every entity that should be seen by the user*: The \brief field is used to provide the short one-line documentation string that is included in the function summary section of classes, groups, namespaces etc. Even if no text documentation is needed/wanted, add an empty

```
/** \brief . */
void someFunction();
```

```
/** \brief Apply the linear operator to a multi-vector : <tt>Y =
 * alpha*op(M)*X + beta*Y</tt>.
 *
 * \param M_trans [in] Determines whether the operator is applied or the
 * adjoint for <tt>op(M)</tt>.
 *
 * \param X [in] The right hand side multi-vector.
 *
 * \param Y [in/out] The target multi-vector being transformed.  When
 * <tt>beta==0.0</tt>, this multi-vector can have uninitialized elements.
 *
 * \param alpha [in] Scalar multiplying <tt>M</tt>, where <tt>M==*this</tt>.
 * The default value of <tt>alpha</tt> is </tt>1.0</tt>
 *
 * \param beta [in] The multiplier for the target multi-vector <tt>Y</tt>.
 * The default value of <tt>beta</tt> is <tt>0.0</tt>.
 *
 * <b>Preconditions:</b><ul>
 *
 * <li> <tt>nonnull(this->domain()) && nonnull(this->range())</tt>
 *
 * <li> <tt>this->opSupported(M_trans)==true</tt> (throw
 * <tt>Exceptions::OpNotSupported</tt>)
 *
 * <li> <tt>X.range()->isCompatible(*op(this)->domain()) == true</tt> (throw
 * <tt>Exceptions::IncompatibleVectorSpaces</tt>)
 *
 * <li> <tt>Y->range()->isCompatible(*op(this)->range()) == true</tt> (throw
 * <tt>Exceptions::IncompatibleVectorSpaces</tt>)
 *
 * <li> <tt>Y->domain()->isCompatible(*X.domain()) == true</tt> (throw
 * <tt>Exceptions::IncompatibleVectorSpaces</tt>)
 *
 * <li> <tt>Y</tt> can not alias <tt>X</tt>.  It is up to the client to
 * ensure that <tt>Y</tt> and <tt>X</tt> are distinct since in general this
 * can not be verified by the implementation until, perhaps, it is too late.
 * If possible, an exception will be thrown if aliasing is detected.
 *
 * </ul>
 *
 * <b>Postconditions:</b><ul>
 * <li> Is it not obvious?  After the function returns the multi-vector <tt>Y</tt>
 *      is transformed as indicated above.
 * </ul>
 */
void apply(
  const EOpTransp M_trans,
  const MultiVectorBase<Scalar> &X,
  const Ptr<MultiVectorBase<Scalar> > &Y,
  const Scalar alpha,
  const Scalar beta
  ) const;
```

Figure 2. Example of more complete doxygen documentation for a function.

comment so that Doxygen will include the class, function, or other entity in the HTML documentation. Note that this is important when the Doxygen configuration option `EXTRACT_ALL` is set to `NO`.

- **DOX 5**: *Add a `\param` field for all of the arguments or none of the the arguments in a function; do not define partial `\param` field lists*: All arguments should be listed in `\param` fields with at least the [in], [out], or [in/out] specifications and these should have at least a very short description. Or, if the function arguments are clear and trivial (and/or have already been defined in the common documentation section), then no `\param` fields for any of the arguments should be included at all. If any of the arguments in a function's documentation are listed in `\param` fields then all arguments should be listed in `\param` fields.

- **DOX 6**: *Only add a `\returns` field if necessary and if so refer to the return object as `returnVal`*: Don't add a `\returns` description of the return value if it is already clearly specified in the `\brief` description of the function. However, if the nature of the return value is at all complex, then include a `\returns` field to describe it. When referring to the return object, refer to it as `returnVal`. By consistently using the identifier `returnVal` for the return value, user's will immediately know what this is referring to.

- **DOX 7**: *Prefer specifying postconditions for output arguments in their `\param` field; otherwise specify their postconditions in the 'postconditions' list*: The postconditions for output arguments can be listed directly in the `\param` field for the argument if they only involve just that argument in a fairly simple way. Otherwise, if the postconditions are more complex or involve multiple arguments in order to specify, then they can be listed in the postconditions list. It may be difficult to objectively determine the best place to list the postconditions for an output argument.

- **DOX 8**: *Order the documentation fields in function documentation as `\brief`, `\param`, Preconditions, Postconditions, `\returns`, then detailed documentation; omitting those components that do not apply*: A consistent ordering of sections of documentation for a function makes it easier for readers to find what they are looking for.

- **DOX 9**: *If possible, try to use `\relates` to associate nonmember functions with a single class*: If a nonmember function is most closely related to a single class, then use the `\relates` field to cause the documentation for the function to be listed with the classes documentation. This makes it easier for readers to find out everything that they can do with a class object (or set of class objects) just by looking at a single HTML page and a single summary list of functions (which includes member and nonmember related functions).

- **DOX 10**: *Provide detailed documentation for only the initial declaration of a virtual function*: Only provide detailed documentation of the initial declaration of a virtual function in the class where it is first defined as `virtual`. In general, documentation should not be included for the overrides of virtual functions in derived classes. Doxygen automatically puts in a link to the original virtual function in the base class so readers are just one click away for seeing the detailed documentation. Always add an empty

  ```
  /** \brief . */
  void someFunction();
  ```

  comment for every class and every function that should be included in the HTML documentation but where no text documentation is wanted or needed.

- **DOX 11**: *Aggregate the overrides of virtual functions into groups according their base class*: For example, the overrides of the virtual functions for the `Teuchos::ParameterListAcceptor` interface would look like:

```
class SomeClass : public Teuchos::ParameterListAcceptor {
public:

  ...

  /** \name Overriden from Teuchos::ParameterListAccpetor */
  //@{
  /** \brief . */
  void setParameterList(
    Teuchos::RCP<Teuchos::ParameterList> const& paramList);
  /** \brief . */
  Teuchos::RCP<Teuchos::ParameterList> getParameterList();
  /** \brief . */
  Teuchos::RCP<Teuchos::ParameterList> unsetParameterList();
  /** \brief . */
  Teuchos::RCP<const Teuchos::ParameterList> getParameterList() const;
  /** \brief . */
  Teuchos::RCP<const Teuchos::ParameterList> getValidParameters() const;
  //@}

  ...

};
```

- **DOX 12**: *Example source code used in Doxygen-generated and other forms of documentation should be extracted automatically from code that is compiled and tested nightly*: Any significant fragment of example code that is shown in Doxygen HTML documentation or a latex document needs to come from compiled and tested code that can be updated automatically. These C++ code fragments can be selectively inserted automatically into Doxygen documentation using the `\dontinclude` Doxygen command.

- **DOX 13**: *Sample output should be generated automatically from compiled and tested code*: Sample output included in Doxygen documentation should be generated automatically by the test harness code and should be written to files that are included in the source directory. The sample output in these files can then be inserted into the Doxygen HTML documentation automatically using the `\verbinclude` Doxygen command. Similar approaches can also be used for latex documentation.

References

[1] R. A. Bartlett. Teuchos C++ memory management classes, idioms, and related topics: The complete reference (a comprehensive strategy for safe and efficient memory management in C++ for high performance computing). Technical report SAND2010-2234, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2010.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2000.

[3] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[4] T. Davidson and J. Pattee. Artistic style 1.20. `http://astyle.sourceforge.net`.

[5] Lockheed Martin. Joint strike fighter air vehicle c++ coding standards for the system development and demonstration program. Technical report 2RDU00001 Rev C, Lockheed Martin Corporation, 2005.

[6] R. Martin. *Agile Software Development (Principles, Patterns, and Practices).* Prentice Hall, 2003.

[7] S. McConnell. *Code Complete: Second Edition.* Microsoft Press, 2004.

[8] S. Meyers. *Effective C++: Third Edition.* Addison Wesley, 2005.

[9] B. Stroustrup. *The C++ Programming Language, special edition.* Addison-Wesley, New York, 1997.

[10] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines and Best Practices*. Addison Wesley, 2005.

A Summary of guidelines

NC (Naming conventions)

- **NC 1**: *Capitalize C++ class and struct names as* `SomeClass`.
- **NC 2**: *Capitalize C++ namespace names as* `SomeNameSpace`.
- **NC 3**: *C++ enum type names should begin with* `E` *as* `ESomeEnum` *and enum values should use all caps and scope context as* `SOME_ENUM_VALUE`.
- **NC 4**: *C++ object instance identifier names should begin with a lower-case letter as* `someObject`.
- **NC 5**: *C++ class data member names should begin with a lower-case letter and end with an underscore as* `someDataMember_`
- **NC 6**: *C++ function names should begin with a lower-case letter as* `someFunction(...)`.
- **NC 7**: *Name C++ pure abstract base classes* `BlobBase`*, default implementation base classes* `BlobDefaultBase`*, and default concrete implementation classes* `DefaultTypeABlob`.
- **NC 8**: *Prefer to name const and non-const access functions as* `getPart()` *and* `getNonconstPart()`*, respectively.*

NOSF (Naming and organization of source files)

- **NOSF 1**: *Use file extension names* `*.hpp` *(C++ header),* `*.cpp` *(C++ source),* `*.h` *(C header), and* `*.c` *(C source).*
- **NOSF 2**: *Include only one major C++ class with supporting code per header and source file with name(s)* `NameSpaceA_InnerNamespace_SomeClass.[hpp,cpp]`.
- **NOSF 3**: *Use internal include guards in all header files.*

GCG (General coding guidelines)

- **Error handling**
  - **GCG 1**: *Use* `TEST_FOR_EXCEPTION(...)`, `TEUCHOS_ASSERT(...)` *and related macros for reporting all errors, even developer programming errors.*
- **Memory management**
  - **GCG 2**: *Avoid the use of raw C++ pointers in all but the very specialized siturations.*
  - **GCG 3**: *Use* `std::string` *instead of* `char*` *or* `const char*`
  - **GCG 4**: *Use* `Teuchos::Ptr` *as function arguments and return types in the place of raw C++ pointers to single objects for non-persisting and semi-persisting associations.*

  - **GCG 5**: *Use* `Teuchos::RCP` *for memory management of single dynamically allocated objects and for handling persisting associations.*
  - **GCG 6**: *Use non-member constructors for all reference-type classes to force dynamic allocation returning strong owning* `Teuchos::RCP` *objects.*
  - **GCG 7**: *Specify "generalized view" semantics for all views of abstract objects.*
  - **GCG 8**: *Use* `Teuchos::ArrayView` *as function arguments and return types in the place of pointers into raw arrays or other container classes for non-persisting and semi-persisting associations and where the array does not need to be resized.*
  - **GCG 9**: *Use* `Teuchos::Array` *in place of* `std::vector` *as a contiguous general purpose data container.*
  - **GCG 10**: *Use* `Teuchos::ArrayRCP` *for memory management of dynamically allocated objects stored in contiguous arrays of data and for persisting associations involving contiguous arrays.*
  - **GCG 11**: *Always return* `Ptr`, `RCP`, `ArrayView`, *and* `ArrayRCP` *smart pointer objects by value, never by reference.*
  - **GCG 12**: *Only return a raw C++ reference from a function for non-persisting associaitons and use the reference and discard it in the same same statement.*
  - **GCG 13**: *Return only* `Ptr` *and* `ArrayView` *objects by value to establish semi-persisting associations; never use a raw C++ reference for a semi-persisting association.*
  - **GCG 14**: *When raw C++ pointers must be exposed (i.e., due to interfacing with non-compliant code), minimize the amount of code exposed to the raw pointer.*

- **Object Control**
  - **GCG 15**: *Accept user options at runtime through a* `Teuchos::ParameterList` *object by deriving from the* `Teuchos::ParameterListAcceptor` *interface.*
  - **GCG 16**: *Fully validate all parameters and sublists in accepted* `Teuchos::ParameterList` *objects using* `validatePamaters(...)` *and other means.*

- **Object Introspection**
  - **GCG 17**: *Always send output to some general* `std::ostream` *object; Never send output directly to* `std::cout` *or* `std::cerr`*; Never print output with* `print(...)` *or* `printf(...)`*.*
    * *Prefer to print output through a* `Teuchos::FancyOStream` *object instead of through a bare* `std::ostream` *object to more easily produce indented formatted output.*
    * *Derive from* `Teuchos::Describable` *and implement the functions* `description()` *and* `describe()` *to allow clients to print the current state of an object.*
    * *Derive from* `Teuchos::VerboseObject` *and print to* `*this->getOStream()` *to give information about what an object is doing.*
    * *As a last resort, always prefer printing to* `*Teuchos::VerboseObjectBase::getDefaultOStream()` *instead of* `std::cout` *or* `std::cerr`*.*

- **Miscellaneous coding guidelines**
  - **GCG 18**: *Prefer to explicitly specify template arguments in a template function call to avoid protability problems and enable implicit covnersions of input arguments.*
  - **GCG 19**: *Use the template function* `Teuchos::as<T_to>(T_from)` *for all conversion of value data types that may result in loss of precision or in an incorrect conversion.*
  - **GCG 20**: *Use namespace enclosure for the definition of C++ class members.*
  - **GCG 21**: *Use explicit namespace qualification for the definition of all nonmember C++ functions.*
  - **GCG 22**: *For general functions, prefer to list function arguments in the order of input, input/output, output, and finally optional arguments with default values.*
  - **GCG 23**: *For non-member object functions, list the object as the first argument passed in as a const reference or non-const reference.*
  - **GCG 24**: *Prefer* `enum`*s to* `bool`*s as formal function arguments when conversion mistakes are likely.*
  - **GCG 25**: *Avoid overloading virtual functions.*
  - **GCG 26**: *Avoid overloading functions on different smart pointer types (e.g.,* `RCP`*,* `Ptr`*, etc.).*
  - **GCG 27**: *Include only standard C++ headers* `<cX>`*, not standard C headers* `<X.h>`*, and avoid all* `using namespace std` *directives.*
  - **GCG 28**: *Break up templated code into four files* `SomeClass_decl.hpp`*,* `SomeClass_def.hpp`*,* `SomeClass.hpp`*, and* `SomeClass.cpp` *to support both implicit and explicit instantiation, minimize recompilation, and avoid problems in mutually dependent (i.e. circular) declarations.*

FSCP (General principles for formatting of source code)

- **FSCP 1**: *Formatting should accurately and consistently show the logical structure of the code.*
- **FSCP 2**: *Formatting should improve the readability of the code for most people.*
- **FSCP 3**: *Formatted code should retain its formatting well when modified; especially for those modifications performed by automated tools.*
- **FSCP 4**: *Formatting style should follow the most common idiom unless one of the above principles are violated.*

FSC (Specific source code formatting principles)

- **FSC 1**: *The formatting style in any single file or group of closely related files should be the same.*
- **FSC 2**: *Try to keep all text within the first 80 character columns.*
- **FSC 3**: *Indent with spaces and not tabs (two spaces by default).*
- **FSC 4**: *Use two vertical spaces to separate class declarations, function definitions, namespace enclosure bounds, and other such major entries in a file.*

- **FSC 5**: *Do not indent source code inside of namespace enclosures, instead use commented end braces.*
- **FSC 6**: *C++ class declarations should generally be laid out with* `public` *members coming before* `protected` *members coming before* `private` *members and indented as shown in Figure 1.*
- **FSC 7**: *List short function prototypes on one line and longer prototypes on multiple lines, indenting arguments one unit.*
  - *List short function prototypes on one line if possible.*
  - *For longer prototypes, indent arguments on continuation lines one unit.*
  - *Return types can be listed on same line as the function name unless the line is too long.*
- **FSC 8**: *Order the definitions of C++ entities the same as the order of the declarations of those entities.*
- **FSC 9**: *Use "modified K&R" or "ANSI" style for the placement of braces and indentation of control structures.*

DOXP (Goals for function and class level documentation)

- **DOXP 1**: *The level of documentation should vary depending on the prominence and/or the role of the software entity or collection.*
- **DOXP 2**: *Important abstract interfaces must be fully specified independent of any single concrete implementation (i.e. preconditions, postconditions, invariants, etc.).*
- **DOXP 3**: *Behavior of "user level" interfaces must be completely specified by the Doxygen documentation and/or other higher-level documentation (i.e. preconditions, postconditions, invariants, etc.).*
- **DOXP 4**: *Wrong documentation is (almost) worse than no documentation at all.*
- **DOXP 5**: *The same documentation should not be repeated in more than one place if possible.*
- **DOXP 6**: *The documentation should maintain itself as much as possible and be testable as much as possible.*

DOX (General Doxygen documentation principles)

- **DOX 1**: *Write Doxygen documentation directly in header files with documented entities.*
- **DOX 2**: *Use a centralized set of definitions for common arguments whenever possible.*
- **DOX 3**: *Provide typical pre- and postconditions along with the documentation for common arguments whenever possible.*
- **DOX 4**: *Add a* `\brief` *description for every entity that should be seen by the user.*

- **DOX 5**: *Add a* `\param` *field for all of the arguments or none of the the arguments in a function; do not define partial* `\param` *field lists.*
- **DOX 6**: *Only add a* `\returns` *field if necessary and if so refer to the return object as* `returnVal`.
- **DOX 7**: *Prefer specifying postconditions for output arguments in their* `\param` *field; otherwise specify their postconditions in the 'postconditions' list.*
- **DOX 8**: *Order the documentation fields in function documentation as* `\brief`*,* `\param`*, Preconditions, Postconditions,* `\returns`*, then detailed documentation; omitting those components that do not apply.*
- **DOX 9**: *If possible, try to use* `\relates` *to associate nonmember functions with a single class.*
- **DOX 10**: *Provide detailed documentation for only the initial declaration of a virtual function.*
- **DOX 11**: *Aggregate the overrides of virtual functions into groups according their base class.*
- **DOX 12**: *Example source code used in Doxygen-generated and other forms of documentation should be extracted automatically from code that is compiled and tested nightly.*
- **DOX 13**: *Sample output should be generated automatically from compiled and tested code.*

B Summary of Teuchos memory management classes and idioms

Basic Teuchos smart pointer types

	Non-persisting (and semi-persisting) Associations	Persisting Associations
single objects	`Ptr<T>`	`RCP<T>`
contiguous arrays	`ArrayView<T>`	`ArrayRCP<T>`

Other Teuchos array container classes

Array class	Specific use case
`Array<T>`	Contiguous dynamically sizable, expandable, and contractible arrays
`Tuple<T,N>`	Contiguous statically sized (with size `N`) arrays

Equivalencies for const protection for raw pointers and Teuchos smart pointers types

Description	Raw pointer	Smart pointer
Basic declaration (non-const obj)	`typedef A* ptr_A`	`RCP<A>`
Basic declaration (const obj)	`typedef const A* ptr_const_A`	`RCP<const A>`
non-const pointer, non-const object	`ptr_A`	`RCP<A>`
const pointer, non-const object	`const ptr_A`	`const RCP<A>`
non-const pointer, const object	`ptr_const_A`	`RCP<const A>`
const pointer, const object	`const ptr_const_A`	`const RCP<const A>`

Summary of operations supported by the basic Teuchos smart pointer types

Operation	`Ptr<T>`	`RCP<T>`	`ArrayView<T>`	`ArrayRCP<T>`
Raw pointer-like functionality				
Implicit conv derived to base	x	x		
Implicit conv non-const to const	x	x	x	x
Dereference `operator*()`	x	x		x
Member access `operator->()`	x	x		x
`operator[](i)`			x	x
operators `++`, `--`, `+=(i)`, `-=(i)`				x
Other functionality				
Reference counting machinery		x		x
Iterators: begin(), end()			x	x
`ArrayView` subviews			x	x

Basic implicit and explicit supported conversions for Teuchos smart pointer types

Operation	`Ptr<T>`	`RCP<T>`	`ArrayView<T>`	`ArrayRCP<T>`
Implicit conv derived to base	x	x		
Implicit conv non-const to const	x	x	x	x
`const_cast`	x	x	x	x
`static_cast`	x	x		
`dynamic_cast`	x	x		
`reinterpret_cast`			x	x

Class Data Members for Value-Type Objects

Data member purpose	Data member declaration
non-shared, single, const object	`const S s_;`
non-shared, single, non-const object	`S s_;`
non-shared array of non-const objects	`Array<S> as_;`
shared array of non-const objects	`RCP<Array<S> > as_;`
non-shared statically sized array of non-const objects	`Tuple<S,N> as_;`
shared statically sized array of non-const objects	`RCP<Tuple<S,N> > as_;`
shared fixed-sized array of const objects	`ArrayRCP<const S> as_;`
shared fixed-sized array of non-const objects	`ArrayRCP<S> as_;`

Class Data Members for Reference-Type Objects

Data member purpose	Data member declaration
non-shared or shared, single, const object	`RCP<const A> a_;`
non-shared or shared, single, non-const object	`RCP<A> a_;`
non-shared array of const objects	`Array<RCP<const A> > aa_;`
non-shared array of non-const objects	`Array<RCP<A> > aa_;`
shared fixed-sized array of const objects	`ArrayRCP<RCP<const A> > aa_;`
"..." (const ptr)	`ArrayRCP<const RCP<const A> > aa_;`
shared fixed-sized array of non-const objects	`ArrayRCP<RCP<const A> > aa_;`
"..." (const ptr)	`ArrayRCP<const RCP<const A> > aa_;`

Passing IN Non-Persisting Associations to Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object (required)	`const A &a`
single, non-changeable object (optional)	`const Ptr<const A> &a`
single, changeable object (required)	`const Ptr<A> &a` **or** `A &a`
single, changeable object (optional)	`const Ptr<A> &a`
array of non-changeable objects	`const ArrayView<const Ptr<const A> > &aa`
array of changeable objects	`const ArrayView<const Ptr<A> > &aa`

Passing IN Persisting Associations to Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const RCP<const A> &a`
single, changeable object	`const RCP<A> &a`
array of non-changeable objects	`const ArrayView<const RCP<const A> > &aa`
array of changeable objects	`const ArrayView<const RCP<A> > &aa`

Passing OUT Persisting Associations for Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const Ptr<RCP<const A> > &a`
single, changeable object	`const Ptr<RCP<A> > &a`
array of non-changeable objects	`const ArrayView<RCP<const A> > &aa`
array of changeable objects	`const ArrayView<RCP<A> > &aa`

Passing OUT Semi-Persisting Associations for Reference (or Value) Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object	`const Ptr<Ptr<const A> > &a`
single, changeable object	`const Ptr<Ptr<A> > &a`
array of non-changeable objects	`const ArrayView<Ptr<const A> > &aa`
array of changeable objects	`const ArrayView<Ptr<A> > &aa`

Passing IN Non-Persisting Associations to Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
single, non-changeable object (required)	`S s` **or** `const S s` **or** `const S &s`
single, non-changeable object (optional)	`const Ptr<const S> &s`
single, changeable object (required)	`const Ptr<S> &s` **or** `S &s`
single, changeable object (optional)	`const Ptr<S> &s`
array of non-changeable objects	`const ArrayView<const S> &as`
array of changeable objects	`const ArrayView<S> &as`

Passing IN Persisting Associations to Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const ArrayRCP<const S> &as`
array of changeable objects	`const ArrayRCP<S> &ss`

Passing OUT Persisting Associations for Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const Ptr<ArrayRCP<const S> > &as`
array of changeable objects	`const Ptr<ArrayRCP<S> > &as`

Passing OUT Semi-Persisting Associations for Value Objects as Func Args

Argument Purpose	Formal Argument Declaration
array of non-changeable objects	`const Ptr<ArrayView<const S> > &as`
array of changeable objects	`const Ptr<ArrayView<S> > &as`

Returning Non-Persisting Associations to Value Objects

Purpose	Return Type Declaration
Single copied object (return by value)	`S`
Single non-changeable object (required)	`const S&`
Single non-changeable object (optional)	`Ptr<const S>`
Single changeable object (required)	`S&`
Single changeable object (optional)	`Ptr<S>`
Array of non-changeable objects	`ArrayView<const S>`
Array of changeable objects	`ArrayView<S>`

Returning Persisting Associations to Value Objects

Purpose	Return Type Declaration
Array of non-changeable objects	`ArrayRCP<const S>`
Array of changeable objects	`ArrayRCP<S>`

Returning Semi-Persisting Associations to Value Objects

Purpose	Return Type Declaration
Array of non-changeable objects	`ArrayView<const S>`
Array of changeable objects	`ArrayView<S>`

Returning Non-Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single cloned object	`RCP<A>`
Single non-changeable object (required)	`const A&`
Single non-changeable object (optional)	`Ptr<const A>`
Single changeable object (required)	`A&`
Single changeable object (optional)	`Ptr<A>`
Array of non-changeable objects	`ArrayView<const Ptr<const A> >`
Array of changeable objects	`ArrayView<const Ptr<A> >`

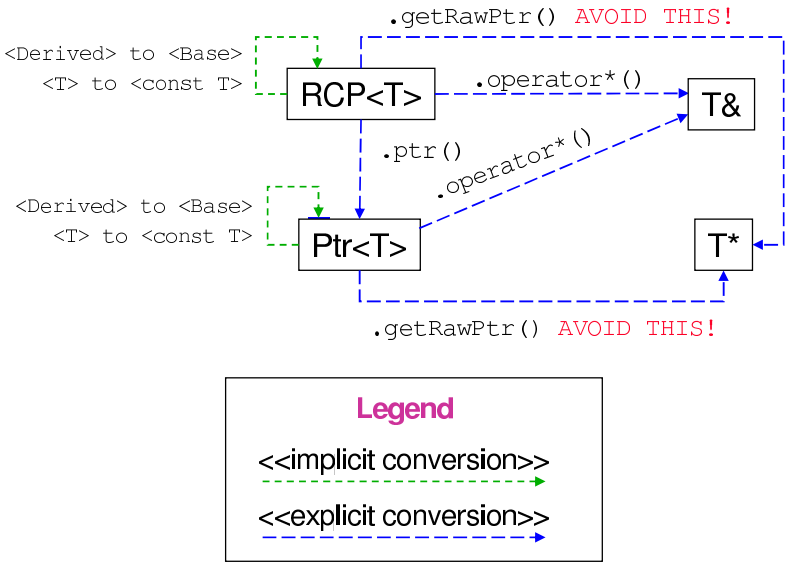
Returning Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single non-changeable object	`RCP<const A>`
Single changeable object	`RCP<A>`
Array of non-changeable objects	`ArrayView<const RCP<const A> >`
Array of changeable objects	`ArrayView<const RCP<A> >`

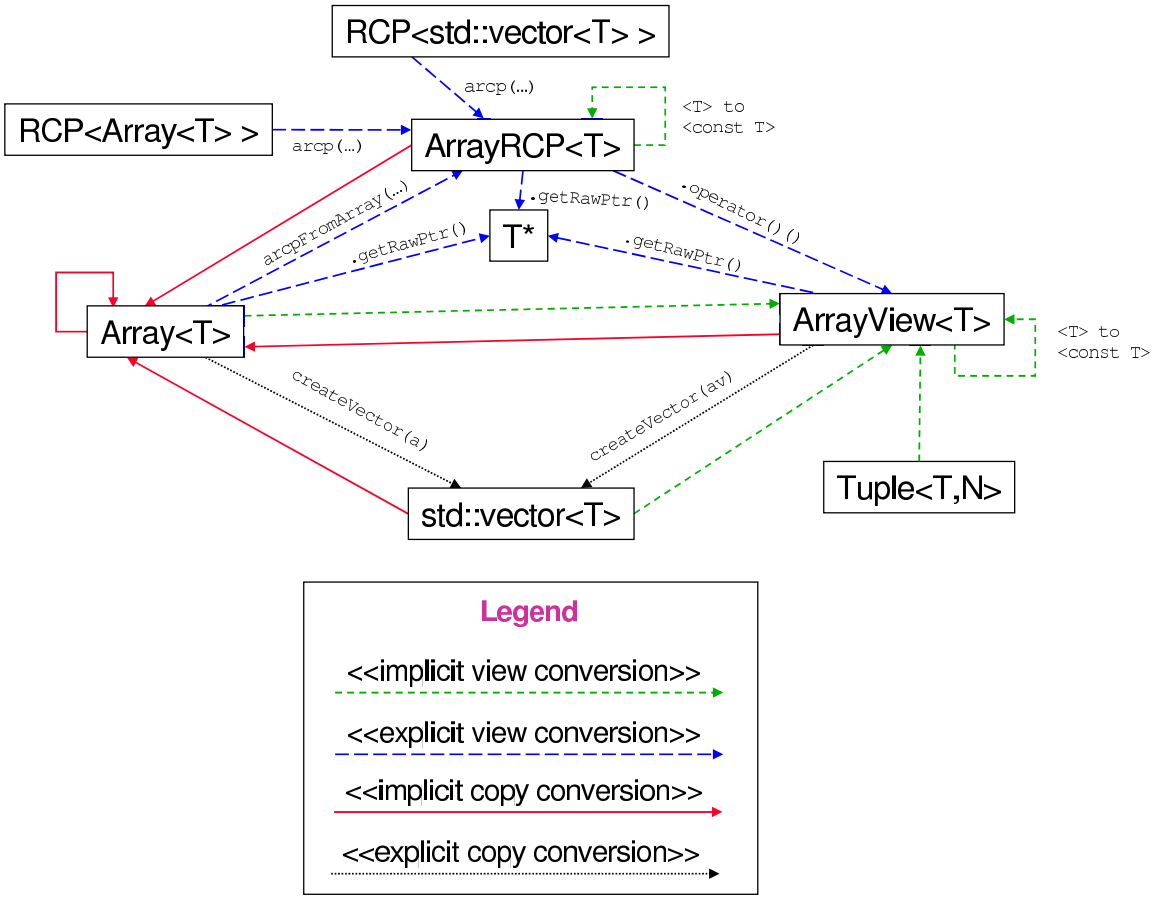
Returning Semi-Persisting Associations to Reference (or Value) Objects

Purpose	Return Type Declaration
Single non-changeable object	`Ptr<const A>`
Single changeable object	`Ptr<A>`
Array of non-changeable objects	`ArrayView<const Ptr<const A> >`
Array of changeable objects	`ArrayView<const Ptr<A> >`

Conversions of data-types for single objects



Conversions of data-types for contiguous arrays



Most Common Basic Conversions for Single Object Types

Type To	Type From	Properties	C++ code
RCP<A>	A*	Ex, Ow	rcp(a_p) [1]
RCP<A>	A*	Ex, NOw	rcp(a_p,false) [2]
RCP<A>	A&	Ex, NOw	rcpFromRef(a)
RCP<A>	A&	Ex, NOw	rcpFromUndefRef(a)
RCP<A>	Ptr<A>	Ex, NOw, DR	rcpFromPtr(a)
RCP<A>	boost::shared_ptr<A>	Ex, Ow, DR	rcp(a_sp)
RCP<const A>	RCP<A>	Im, Ow, DR	RCP<const A>(a_rcp)
RCP<Base>	RCP<Derived>	Im, Ow, DR	RCP<Base>(derived_rcp)
RCP<const Base>	RCP<Derived>	Im, Ow, DR	RCP<const Base>(derived_rcp)
boost::shared_ptr<A>	RCP<A>	Ex, Ow, DR	shared_pointer(a_rcp)
A*	RCP<A>	Ex, NOw	a_rcp.getRawPtr() [3]
A&	RCP<A>	Ex, NOw	*a_rcp [4]
Ptr<A>	A*	Ex, NOw	ptr(a_p) [2]
Ptr<A>	A&	Ex, NOw	outArg(a) [5]
Ptr<A>	RCP<A>	Ex, NOw, DR	a_rcp.ptr()
Ptr<A>	RCP<A>	Ex, NOw, DR	a_rcp()
Ptr<A>	RCP<A>	Ex, NOw, DR	ptrFromRCP(a_rcp)
Ptr<const A>	Ptr<A>	Im, NOw, DR	Ptr<const A>(a_ptr)
Ptr<Base>	Ptr<Derived>	Im, NOw, DR	Ptr<Base>(derived_ptr)
Ptr<const Base>	Ptr<Derived>	Im, NOw, DR	Ptr<const Base>(derived_ptr)
A*	Ptr<A>	Ex, NOw	a_ptr.getRawPtr() [3]
A&	Ptr<A>	Ex, NOw	*a_ptr() [4]
A*	A&	Ex, NOw	&a [3]
A&	A*	Ex, NOw	*a_p [3]

Types/identifiers: `A* a_p`; `A& a`; `Ptr<A> a_ptr`; `RCP<A> a_rcp`; `boost::shared_ptr<A> a_sp`;

Properties: Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning, NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection (NOTE: All conversions are shallow conversions, i.e. copies pointers not objects.)

1. Constructing an owning `RCP` from a raw C++ pointer is strictly necessary but must be done with great care according to the commandments in Appendix **??**.
2. Constructing a non-owning `RCP` or `Ptr` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer and raw pointer manipulation should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section **??**).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.
5. See other helper constructors for passing `Ptr` described in Section **??**.

Most Common Basic Conversions for Contiguous Array Types

Type To	Type From	Properties	C++ code (or impl function)
`ArrayRCP<S>`	`S*`	Sh, Ex, Ow	`arcp(s_p,0,n)` [1]
`ArrayRCP<S>`	`S*`	Sh, Ex, NOw	`arcp(s_p,0,n,false)` [2]
`ArrayRCP<S>`	`Array<S>`	Sh, Ex, NOw, DR	`arcpFromArray(s_a)`
`ArrayRCP<S>`	`ArrayView<S>`	Sh, Ex, NOw, DR	`arcpFromArrayView(s_av)`
`ArrayRCP<S>`	`ArrayView<S>`	Dp, Ex, Ow	`arcpClone(s_av)`
`ArrayRCP<S>`	`RCP<Array<S> >`	Sh, Ex, Ow, DR	`arcp(s_a_rcp)`
`ArrayRCP<const S>`	`RCP<const Array<S> >`	Sh, Ex, Ow, DR	`arcp(cs_a_rcp)`
`ArrayRCP<const S>`	`ArrayRCP<S>`	Sh, Im, Ow, DR	`ArrayRCP::operator()()`
`S*`	`ArrayRCP<S>`	Sh, Ex, NOw	`s_arcp.getRawPtr()` [3]
`S&`	`ArrayRCP<S>`	Sh, Ex, NOw	`s_arcp[i]` [4]
`ArrayView<S>`	`S*`	Sh, Ex, NOw	`arrayView(s_p,n)` [1]
`ArrayView<S>`	`Array<S>`	Sh, Im, NOw, DR	`Array::operator ArrayView()`
`ArrayView<S>`	`Tuple<S>`	Sh, Im, NOw, DR	`Tuple::operator ArrayView()`
`ArrayView<S>`	`std::vector<S>`	Sh, Im, NOw	`ArrayView<S>(s_v)`
`ArrayView<S>`	`ArrayRCP<S>`	Sh, Ex, NOw, DR	`ArrayRCP::operator()()`
`ArrayView<const S>`	`const Array<S>`	Sh, Im, NOw, DR	`Array::operator ArrayView()`
`ArrayView<const S>`	`const Tuple<S>`	Sh, Im, NOw, DR	`Tuple::operator ArrayView()`
`ArrayView<const S>`	`const std::vector<S>`	Sh, Im, NOw	`ArrayView(cs_v)`
`ArrayView<const S>`	`ArrayRCP<const S>`	Sh, Ex, NOw, DR	`ArrayRCP::operator ArrayView()`
`S*`	`ArrayView<S>`	Ex, NOw	`s_av.getRawPtr()` [3]
`S&`	`ArrayView<S>`	Ex, NOw	`s_av[i]` [4]
`Array<S>`	`S*`	Dp, Ex	`Array<S>(s_p,s_p+n)`
`Array<S>`	`std::vector<S>`	Dp, Im	`Array<S>(s_v)`
`Array<S>`	`ArrayView<S>`	Dp, Im	`Array<S>(s_av)`
`Array<S>`	`Tuple<S,N>`	Dp, Im	`Array<S>(s_t)`
`Array<S>`	`ArrayRCP<S>`	Dp, Ex	`Array<S>(s_arcp());`
`std::vector<S>`	`Array<S>`	Dp, Ex	`s_a.toVector();`
`S*`	`Array<S>`	Ex, NOw	`s_a.getRawPtr()` [3]
`S&`	`Array<S>`	Ex, NOw	`s_a[i]` [4]

Types/identifiers: `S* s_p; ArrayView<S> s_av; ArrayRCP<S> s_arcp; Array<S> s_a; Tuple<S,N> s_t; std::vector<S> s_v; RCP<Array<S> > s_a_rcp; RCP<const Array<S> > cs_a_rcp;`

Properties: Sh = Shallow copy, Dp = Deep copy (dangling references not an issue), Im = Implicit conversion, Ex = Explicit conversion, Ow = Owning (dangling references not an issue), NOw = Non-Owning, DR = Dangling Reference debug-mode runtime detection for non-owning

1. It should never be necessary to convert from a raw pointer to an owning `ArrayRCP` object directly. Instead, use the non-member constructor `arcp<S>(n)`.
2. Constructing a non-owning `ArrayRCP` or `ArrayView` directly from a raw C++ pointer should never be needed in fully compliant code. However, when inter-operating with non-compliant code (or code in an intermediate state of refactoring) this type of conversion will be needed.
3. Exposing a raw C++ pointer should never be necessary in compliant code but may be necessary when inter-operating with external code (see Section **??**).
4. Exposing a raw C++ reference will be common in compliant code but should only be used for non-persisting associations.

C Summary of "C++ Coding Standards" (CPPCS) with amendments

Below, the 101 items in "C++ Coding Standards" by Sutter and Alexandrescu [10] are listed along with items that are amended or invalidated in the Thyra coding guidelines. General amendments that apply to all items are:

- Replace `tr1::shared_ptr` with `Teuchos::RCP`
- Replace `std::vector` with `Teuchos::Array`
- Replace `assert(someTest)` with `TEUCHOS_ASSERT(someTest)`

Organizational and Policy Issues :

Item 0 : Don't sweat the small stuff. (Or: Know what not to standardize.)
[**Amended**, see Section 6 and Appendix E]

Item 1 : Compile cleanly at high warning levels

Item 2 : Use an automated build system.

Item 3 : Use a version control system.

Item 4 : Invest in code reviews

Design Style :

Item 5 : Give one entity one cohesive responsibility.

Item 6 : Correctness, simplicity, and clarity come first.

Item 7 : Know when and how to code for scalability.

Item 8 : Don't optimize prematurely.

Item 9 : Don't pessimize prematurely.

Item 10 : Minimize global and shared data.

Item 11 : Hide information.

Item 12 : Know when and how to code for concurrency.

Item 13 : Ensure resources are owned by objects. Use explicit RAII and smart pointers.

Coding Style :

Item 14 : Prefer compile- and link-time errors to run-time errors.

Item 15 : Use const proactively.

Item 16 : Avoid macros.

Item 17 : Avoid magic numbers.

Item 18 : Declare variables as locally as possible.

Item 19 : Always initialize variables.

Item 20 : Avoid long functions. Avoid deep nesting.

Item 21 : Avoid initialization dependencies across compilation units.

Item 22 : Minimize definitional dependencies. Avoid cyclic dependencies.

Item 23 : Make header files self-sufficient.

Item 24 : Always write internal #include guards. Never write external #include guards

Functions and Operators :

Item 25 : Take parameters appropriately by value, (smart) pointer, or reference.
[**Amended** by Section 5.2]

Item 26 : Preserve natural semantics for overloaded operators.

Item 27 : Prefer the canonical forms of arithmetic and assignment operators.

Item 28 : Prefer the canonical form of ++ and –. Prefer calling the prefix forms.

Item 29 : Consider overloading to avoid implicit type conversions.

Item 30 : Avoid overloading '&&', '||', or ',' (comma).

Item 31 : Don't write code that depends on the order of evaluation of function arguments.

Class Design and Inheritance :

Item 32 : Be clear what kind of class you're writing.

Item 33 : Prefer minimal classes to monolithic classes.

Item 34 : Prefer composition to inheritance.

Item 35 : Avoid inheriting from classes that were not designed to be base classes.

Item 36 : Prefer providing abstract interfaces.

Item 37 : Public inheritance is substitutability. Inherit, not to reuse, but to be reused.

Item 38 : Practice safe overriding.

Item 39 : Consider making virtual functions nonpublic, and public functions nonvirtual.

Item 40 : Avoid providing implicit conversions.

Item 41 : Make data members private, except in behaviorless aggregates (C-style structs).

Item 42 : Don't give away your internals.

Item 43 : Pimpl judiciously.

Item 44 : Prefer writing nonmember nonfriend functions.

Item 45 : Always provide new and delete together.

Item 46 : If you provide any class-specific new, provide all of the standard forms (plain, in-place, and nothrow).

Construction, Destruction, and Copying :

Item 47 : Define and initialize member variables in the same order.

Item 48 : Prefer initialization to assignment in constructors.

Item 49 : Avoid calling virtual functions in constructors and destructors.

Item 50 : Make base class destructors public and virtual, or protected and nonvirtual.

Item 51 : Destructors, deallocation, and swap never fail.

Item 52 : Copy and destroy consistently.

Item 53 : Explicitly enable or disable copying.

Item 54 : Avoid slicing. Consider Clone instead of copying in base classes.

Item 55 : Prefer the canonical form of assignment.

Item 56 : Whenever it makes sense, provide a no-fail swap (and provide it correctly).

Namespaces and Modules :

Item 57 : Keep a type and its nonmember function interface in the same namespace.

Item 58 : Keep types and functions in separate namespaces unless they are specifically intended to work together.

Item 59 : Don't write namespace usings in a header file or before an #include.
[**Amended**, see Appendix D]

Item 60 : Avoid allocating and deallocating memory in different modules.
[**Invalidated**, see Appendix D]

Item 61 : Don't define entities with linkage in a header file.

Item 62 : Don't allow exceptions to propagate across module boundaries.
[**Invalidated**, see Appendix D]

Item 63 : Use sufficiently portable types in a module's interface.
[**Invalidated**, see Appendix D]

Templates and Genericity :

Item 64 : Blend static and dynamic polymorphism judiciously.

Item 65 : Customize intentionally and explicitly.

Item 66 : Don't specialize function templates.

Item 67 : Don't write unintentionally nongeneric code.

Error Handling and Exceptions :

Item 68 : Assert liberally to document internal assumptions and invariants

Item 69 : Establish a rational error handling policy, and follow it strictly.

Item 70 : Distinguish between errors and non-errors.

Item 71 : Design and write error-safe code.

Item 72 : Prefer to use exceptions to report errors.

Item 73 : Throw by value, catch by reference.

Item 74 : Report, handle, and translate errors appropriately.

Item 75 : Avoid exception specifications.

STL: Containers :

Item 76 : Use vector by default. Otherwise, choose an appropriate container.

Item 77 : Use vector and string instead of arrays.

Item 78 : Use vector (and `string::c_str`) to exchange data with non-C++ APIs.

Item 79 : Store only values and smart pointers in containers.

Item 80 : Prefer `push_back` to other ways of expanding a sequence.

Item 81 : Prefer range operations to single-element operations.

Item 82 : Use the accepted idioms to really shrink capacity and really erase elements.

STL: Algorithms :

Item 83 : Use a checked STL implementation.
[**Amended**, With GCC, configure Trilinos with `Trilinos_ENABLE_CHECKED_STL=ON`]

Item 84 : Prefer algorithm calls to handwritten loops.

Item 85 : Use the right STL search algorithm.

Item 86 : Use the right STL sort algorithm.

Item 87 : Make predicates pure functions.

Item 88 : Prefer function objects over functions as algorithm and comparer arguments.

Item 89 : Write function objects correctly.

Type Safety :

Item 90 : Avoid type switching; prefer polymorphism.

Item 91 : Rely on types, not on representations.

Item 92 : Avoid using `reinterpret_cast`.

Item 93 : Avoid using `static_cast` on pointers.

Item 94 : Avoid casting away const.

Item 95 : Don't use C-style casts.

Item 96 : Don't memcpy or memcmp non-PODs.

Item 97 : Don't use unions to reinterpret representation.

Item 98 : Don't use varargs (ellipsis).

Item 99 : Don't use invalid objects. Don't use unsafe functions.

Item 100 : Don't treat arrays polymorphically.

D Miscellaneous amendments to “C++ Coding Standards”

In this appendix, some of the amendments mentioned in Appendix C to some of the items in [10] are discussed in more detail.

D.1 Amendments to items related to compiler/linker incompatibilities

There are three items in [10] that relate to portability problems associated with mixing and matching code in different binary libraries compiled with different C++ compilers or with different compiler options. In this context, the authors use the term “module” to mean a single library or a set of libraries containing simiarly compiled binary object code.

In general, one can not assume that object code compiled by two or more different C++ compilers will work together since the name-mangling needed for type-safe linkage is not even specified by the ISO C++ standard. A more typical problem is when the same compiler is used, but different compiler and/or linker options are used. For example, some compilers allow you to turn support for exception handling on and off and if an exception is thrown by one module it will not be handled correctly by another module that has exception handling support turned off. A similar problem can happen when mixing static and shared libraries, in Linux for example, where RTTI is handled differently and can result in dynamic casting failures in cases where it would otherwise succeed.

In our model of software deployment, we distribute source code and a build process that users can manipulate in order to set the exact compiler and linker options to match what is used by other libraries and the application code that uses the libraries. Because we develop class libraries, it is simply not realistic to isolate this type of code into libraries with small “Facade” type interfaces that are advocated in [10].

The specific items that we consider inappropriate are:

- *Item 60: Avoid allocating and deallocating memory in different modules*:
- *Item 62: Don’t allow exceptions to propagate across module boundaries*:
- *Item 63: Use sufficiently portable types in a module’s interface*:

All three of these items are related to the problem of mixing code created by different compiler and/or linker options. However, they may also be related to mixed language programming. For example, in order to ensure that your module is the most reusable, you might create a C-compatible interface that allows clients coding in C (and even Fortran 77 in some cases) to call and be called by your module. If mixed language programming is the issue, then a special `extern "C"` interface should be created for the module which will automatically satisfy Items 60, 62, and 63. Note that reference counting machinery in the `RCP` and `ArrayRCP` classes actually solves the problem of calling `new` and `delete` in different modules that is described in Item 60 because the deallocator object that calls `delete` is create and assigned in the same module where `new` is called which guarantees that they are consistent.

D.2 Amendments for 'using' declarations and directives

In [10, Item 59], the authors say to never put 'using' declarations into header files or before `#includes` and that 'using namespace SomeNamespace' directives are perfectly safe for code in source files after all `#includes`. However, we will argue that:

- employing `using` declarations to inject names of C++ classes or enums from one namespace into another is perfectly safe (this is more lax than what is suggested in [10, Item 59])
- employing a `using namespace SomeNameSpace` directive in any context is harmful and should be avoided (this is more restrictive than what is suggested in [10, Item 59]).

However, we agree that employing `using` declarations for nonmember functions is dangerous and is to be avoided because of problems related to overloading and in what order overloads are declared and used (as described in [10, Item 59]) .

Are all using declarations employed in header files dangerous? In [10, Item 59], the authors clearly show that employing 'using' declarations for nonmember functions is dangerous because of overloading. But what about employing 'using' declarations for C++ classes and other types?

To investigate the issues involved, consider the following toy C++ program (in the file NamespaceClassUsingIssues.cpp):

```
//
// Header-like declarations
//

#include <iostream>
#include <cstdlib>

namespace NamespaceA {

template<class T>
class A {
public:
  explicit A( const T& a ) : a_(a) {}
  void print( std::ostream &os ) const { os << "\na="<<a_<<"\n"; }
private:
  T a_;
};

} // namespace NamespaceA

// Add a using declaration to inject 'A' into another namespace
namespace NamespaceB { using NamespaceA::A; }

// Now use the A class without namespace qualification in NamespaceB
namespace NamespaceB {

A<double> foo( std::ostream &os, const A<int> &aa );
```

```
} // namespace NamespaceB

// Create another A class in the global namespace.  With care, we should
// not have any problems with this and our code should not be affected by
// the presence of this class.
template<class T>
class A {
public:
  explicit A( const T& a ) : a_(a)
    { std::cerr << "\nOh no, called ::A::A(...)!\n"; std::exit(1); }
  void print( std::ostream &os ) { os << "\na="<<a_<<"\n"; }
private:
  T a_;
};

// See what happens when you define another class A in NamespaceB which
// conflicts with the using declaration!  This should not be allowed and
// should be caught by the compiler!

#ifdef SHOW_DUPLICATE_CLASS_A

namespace NamespaceB {

template<class T>
class A {
public:
  explicit A( const T& a ) : a_(a)
    { std::cerr << "\nOh no, called ::A::A(...)!\n"; exit(1); }
  void print( std::ostream &os ) { os << "\na="<<a_<<"\n"; }
private:
  T a_;
};

} // namespace NamespaceB

#endif // SHOW_DUPLICATE_CLASS_A


//
// Implementations
//

// Define function in NamespaceB without namespace qualification for class A
NamespaceB::A<double>
NamespaceB::foo( std::ostream &os, const A<int> &aa )
{
  A<double> ab(2.0);
  aa.print(std::cout);
  ab.print(std::cout);
  return ab;
}
// NOTE: Above, we need explicit namespace qualification for the return
// type 'NamespaceB::A<double>' since we use namespace qualification to
// define nonmember functions (see Thyra coding guidelines).  Without this
```

```
// namespace qualification, the global class '::A' would be assumed and
// you would get a compilation error.  However, within the function, which
// is in the scope of NamespaceB, we don't need namespace qualifications!


//
// User's code.  This code does not typically live in a namespace (or is
// in another unrelated namespace).  Here, some explicit namespace
// qualification and using declarations will be required to avoid
// ambiguities.
//

int main()
{

#if defined(SHOW_MISSING_USING_DECL)
  // Here, no using declaration is provided.  This will result in the
  // global class '::A' being used below which will result in a compiler
  // error when the NamespaceB::foo(...) function is called.  This is a
  // feature!
#elif defined(SHOW_ERRONEOUS_USING_DIRECTIVE)
  // Here we try to just inject all of the names from NamespaceA into the
  // local scope.  However, this will result in the names 'NamespaceA::A'
  // and '::A' being equally visible which will result in a compiler error
  // when the first unqualified 'A' object is created below!
  using namespace NamespaceA;
#else
  // Inject the class name 'A' into the local scope and will override any
  // (sloppy) names polluting the global namespace.  This will cause the
  // global '::A' class to be hidden (which is good!).
  using NamespaceA::A;
#endif

  A<int> aa(5);
  A<double> ab = NamespaceB::foo(std::cout,aa);
  ab.print(std::cout);

  return 0;

}
```

The above program defines a templated class `A` in namespace `NamespaceA` and then does a `using NamespaceA::A` to inject this class name into `NamespaceB`.

When the program is compiled and run with g++ (version 4.3.4), one gets:

```
$ g++ -ansi -pedantic -Wall -o NamespaceClassUsingIssues.exe
NamespaceClassUsingIssues.cpp

$ ./NamespaceClassUsingIssues.exe

a=5
```

```
a=2

a=2
```

This program has a few different ifdefs to show different types of errors that a compiler will detect.

1. *What happens if one tries to define another class `A` in namespace `NamespaceB`?*

   In the case of nonmember functions, overloads of a function exhibit strange and non-intuitive behavior when one employs 'using' declarations. However, what happens with classes?

   In the above program, when one defines the macro `SHOW_DUPLICATE_CLASS_A` when compiling, one will get the following compile-time error:

   ```
   $ g++ -ansi -pedantic -Wall -DSHOW_DUPLICATE_CLASS_A \
      -o NamespaceClassUsingIssues.exe NamespaceClassUsingIssues.cpp

   NamespaceClassUsingIssues.cpp:53: error: redefinition of
     'class NamespaceA::A<T>'
   NamespaceClassUsingIssues.cpp:11: error: previous definition of
     'class NamespaceA::A<T>'
   ```

   Above, the error message generated by g++ 4.3.4 is very good and pinpoints the problem exactly. This is in stark contrast to what happens when you have overloaded member functions which [10, Item 59] explains.

   Take-home Message: Employing `using SomeNamespace::SomeClass` declarations to inject names from one namespace into another seems to be safe and does not suffer from the gotchas associated with `using` declarations for (overloaded) nonmember functions.

2. *What happens when the user's code does not have an appropriate using declaration?*

   While the `using NamespaceA::A` declaration in `NamespaceB` allows the code in `NamespaceB` to avoid having to explicitly qualify `NamespaceA::A` all the time, this does not automatically mean that user code that does not live in `NamespaceB` will not have to do something to get at the name `A`. The user can either do explicit qualification `Namespace::A` or can put a `using NamespaceA::A` declaration at the top of their namespace or in each function that they have (as is done in the `main()` function above).

   In the above program, if one defines the macro `SHOW_MISSING_USING_DECL`, the `using Namespace::A` declaration will be missing in `main()` and this will result in the compiler finding the global `::A` class which will cause a compiler error when `NamespaceB::foo(...)` gets called. Here is the error message that one gets when compiling with this macro defined:

   ```
   $ g++ -ansi -pedantic -Wall -DSHOW_MISSING_USING_DECL \
     -o NamespaceClassUsingIssues.exe NamespaceClassUsingIssues.cpp

   NamespaceClassUsingIssues.cpp: In function 'int main()':
   NamespaceClassUsingIssues.cpp:121: error: invalid initialization of
      reference of type 'const NamespaceA::A<int>&' from expression of type '
      A<int>'
   NamespaceClassUsingIssues.cpp:80: error: in passing argument 2 of '
      NamespaceA::A<double> NamespaceB::foo(std::ostream&, const
      NamespaceA::A<int>&)'
   ```

The above error message generated by g++ 4.3.4 here is not all bad as the compiler catches the mistake and states the types involved.

Take-home Message: Always employ `using SomeNamespace::SomeClass` to inject type names from other namespaces that you want to use in your namespace to protect your code from others who pollute the global namespace.

3. *What happens when the user code employs a `using namespace NamespaceA` directive when there are conflicting names?*

   Since there is a global class `::A`, the user can not simply employ a `using namespace NamespaceA` directive or the compiler will complain that it does not know which class to use.

   In the above program, when one defines the macro `SHOW_ERRONEOUS_USING_DIRECTIVE` when compiling one gets the following very good compile error message:

```
$ g++ -ansi -pedantic -Wall -DSHOW_ERRONEOUS_USING_DIRECTIVE \
  -o NamespaceClassUsingIssues.exe NamespaceClassUsingIssues.cpp

NamespaceClassUsingIssues.cpp: In function ‘int main()’:
NamespaceClassUsingIssues.cpp:120: error: use of ‘A’ is ambiguous
NamespaceClassUsingIssues.cpp:45: error:   first declared as ‘
   template<class T> class A’ here
NamespaceClassUsingIssues.cpp:10: error:   also declared as ‘
   template<class T> class NamespaceA::A’ here
NamespaceClassUsingIssues.cpp:120: error: parse error before ‘>’ token
NamespaceClassUsingIssues.cpp:121: error: use of ‘A’ is ambiguous
NamespaceClassUsingIssues.cpp:45: error:   first declared as ‘
   template<class T> class A’ here
NamespaceClassUsingIssues.cpp:10: error:   also declared as ‘
   template<class T> class NamespaceA::A’ here
NamespaceClassUsingIssues.cpp:121: error: parse error before ‘>’ token
NamespaceClassUsingIssues.cpp:122: error: ‘ab’ undeclared (first use
   this function)
NamespaceClassUsingIssues.cpp:122: error: (Each undeclared identifier
   is reported only once for each function it appears in.)
```

   Note that this type of example goes against the advise in [10, Item 59] where the authors state that it is safe to employ `using namespace SomeNamespace` directives in `*.cpp` source files. This example shows that this does not protect the code from others that pollute the global namespace. Note that code that is written this way might compile one day and not the next as it is fragile and can be broken by other people that pollute the global namespace.

   Take-home Message: Never employ `using namespace AnyNamespace` in any context as you cannot guarantee the integrity of your code since people outside of your namespace can cause your code to not compile.

E Arguments for adopting a consistent code formatting style

While there are reasonable ways to handle different code formatting styles within a project (e.g. custom file styles in emacs), there are arguments for preferring a more consistent code formatting style that is used throughout a project by all developers in the project. It is typically more difficult to modify code than to read code that uses an unfamiliar coding style and therefore consistent coding styles is more important in cases where multiple developers modify the same code base.

One of the more lenient opinions on coding style in the literature comes from [10, Item 0] where the authors state:

> "Do use consistent formatting within each source file or even each project, because it's jarring to jump around among several styles in the same piece of code. But don't try to enforce consistent formatting across multiple projects or across a company[7]".

Much stronger arguments for working toward a consistent code formatting style within a project are made by other individuals and organizations who represent a wide range of views of software development. These organizations and persons vary from those associated with open-source organizations (e.g. GNU) to newer Agile methodologists (e.g. Extreme Programming) to large software companies (e.g. Microsoft). As different as these various people and organizations view the nature of software (e.g. GNU vs. Microsoft) and how it should be developed (e.g. GNU vs. Extreme Programming), they all agree that some consistency in coding style is a good idea.

A few points are worth making before looking at opinions on formatting style expressed by these different individuals and organizations. In each of the references cited, the individual or organization gives a justification for the opinions expresses and it is up to the reader to weigh these arguments on their own. Also, just because an opinion is expressed by an "expert" does not in and of itself automatically give that opinion a lot of credence. However, when a wide number of different and diverse "experts" espouse the same opinion, then such a point of view should be considered more seriously.

E.1 Statements on coding style from varied persons and/or organizations

Here we overview some options on consistent code formatting style from a variety of sources.

E.1.1 Open source software (the GNU project)

On one end of the spectrum is the open source software community that one can think of as the freest form of software. A GNU package is usually not even developed by a cohesive set of developers but yet the official GNU Coding Standard[8] states:

[7] The implicit assumption in this latter qualification is that developers don't interact heavily with multiple projects and multiple projects don't interact much with each other and therefore there is typically little advantage to having a company-wide code formatting standard. However, if the same developers work together on multiple projects and go back and forth between projects frequently, it is unclear what the opinion of the authors would be in this case.

[8] `http://www.gnu.org/prep/standards/standards.html`

> "The rest of this section gives our recommendations for other aspects of C formatting style ... We don't think of these recommendations as requirements ... But whatever style you use, please use it consistently, since a mixture of styles within one program tends to look ugly. If you are contributing changes to an existing program, please follow the style of that program".

While the above passage does not mandate a consistent coding style within a GNU package (because it can't, its free software), it does recommend a coding style[9] and it asks that each project please use a consistent coding style throughout a GNU project.

E.1.2 Agile Methods (Extreme Programming)

While the Extreme Programming and GNU movements are miles apart in terms of how it expects coders to work together to create code, they both agree that using a consistent coding style within a project is important.

In his popular 1999 book "Extreme Programming Explained" [2], Kent Beck explicitly listed "Coding Standards" as one of XP's twelve recommended practices. In this book, Beck states

> "You couldn't possibility ask the team to code to a common standard. Programmers are deeply individualistic, and would quit rather than put their curly braces somewhere else. Unless:
>
> - The whole of XP makes them more likely to be members of a winning team.
>
> Then perhaps they could be willing to bend their style a little. Besides, without coding standards the additional friction slows pair programming and refactoring significantly".

In this first book, Beck also comments on coding standards in the context of "collective ownership" of code by stating:

> "You couldn't possibly have everybody potentially changing anything anywhere. Folks would be breaking stuff left and right, and the cost of integration would go up dramatically. Unless:
>
> - You integrate after a short enough time, so that chances of conflicts go down.
> - ...
> - You adhere to coding standards, so you don't get into the dreaded Curly Brace Wars.
>
> Then perhaps you could have anyone change code anywhere in the system when they see the chance to improve it".

As a result, many XP projects have insisted on requiring every member of the team to code in the same way. So much to the point that one should not be able to tell who wrote a piece of code just in how it is formatted. As of this writing, almost every source of information on XP on the Internet takes a very strong opinion on the adoption of a consistent coding style by an XP group. The specific details of the coding

[9] The official GNU formatting style is one of the built-in styles in Emacs called the "gnu" style

style are not important, what is important is that everyone on the team helps to formulate and agrees to use the same coding style.

In his updated 2005 book "Extreme Programming Explained: Second Edition" [3], Kent Beck has restructured XP and now the "Coding Standards" practice is no longer specifically listed as a practice. Does this mean that consistent code formatting is not longer important in XP? The simple answer is no. In her article "The New XP"[10] which outlines the second edition of Beck's book and compares it to the first edition, Michele Marchesi states:

> "You must note that in the new XP we cannot find original practices of *coding standards*, that is considered obvious, ... "

And to put to rest any doubt how Beck himself feels about consistent coding styles he states in the second edition:

> "For example, people get passionate about coding style. While there are undoubtedly better styles and worse styles, the most important style issue is that the team chooses to work towards a common style. Idiosyncratic coding styles and the values revealed by them, individual freedom at all costs, don't help the team succeed".

Therefore, it is clear that the flagship of the Agile programming movement, XP, clearly advocates that a team of developers should work towards a consistent code formatting style.

E.1.3 Code Complete

In [7], Steve McConnell makes a strong argument that groups should adopt a consistent coding standard, including reasonable guidelines for the formatting of source code.

There are several places in his book where McConnell stresses the importance of using a consistent formatting style in a group project:

- "The bottom line is that the details of a specific method of structuring a program are much less important than the fact that the program is structured consistently" [7, Section 31.1]. This quote is almost an exact paraphrase of the statements made in the GNU coding standard document and by Beck in the Extreme Programming books mentioned above.
- "The importance to comprehension and memory of structuring one's environment in a familiarly way has lead some researchers to hypothesize that layout might harm an expert's ability to read a program if the layout is different from the scheme the expert uses (Shell 1981, Soloway and Ehrlich 1984)" [7, Section 31.1]. This implies that working with an unfamiliar style might handicap expert coders more than beginner and intermediate coders.
- "Structuring code is important for its own sake. The specific convention you follow is less important than the fact that you follow the same convention consistently" [7, Chapter 31].

[10] `http://www.agilexp.org/downloads/TheNewXP.pdf`

- "Many aspects of layout are religious issues. Try to separate objective preferences from subjective one. Use explicit criteria to help ground your discussions about style preferences." [7, Chapter 31].
- "Use conventions to spare you brain the challenge of remembering arbitrary differences between different sections of code ." [7, Section 34.1].
- "The point of having coding conventions is to mainly reduce complexity. When you standardized decisions about formatting, loops, variable names, modeling notations, and so on, you release mental resources that you need to focus on more challenging aspects of the programming problem. One reason coding conventions are so controversial is that choices among the options have some limited aesthetic base but are essentially arbitrary. People have the most heated arguments over their smallest differences. Conventions are most useful when they spare you the trouble of making and defending arbitrary decisions. They are less valuable when they impose restrictions in more meaningful areas." [7, Section 34.1].
- "The motivation behind many programming practices is to reduce a program's complexity, and reducing complexity is arguably the most important key to being an effective programmer." [7, Chapter 34].
- "When abused, a programming convention can be a cure that's worse than the disease. Used thoughtfully, a convention adds valuable structure to the development environment and helps with managing complexity and communication." [7, Chapter 34].
- "In general, mandating a strict set of technical standards from the management position isn't a good idea." [7, Section 28.1].
- "If someone on a project is going to define standards, have a respected architect define the standards rather than a manager ... If the architect is regarded as the projects' thought leader, the project team will generally follow standards set by that person." [7, Section 28.1].
- "If your group resists adopting strict standards, consider a few alternatives: flexible guidelines, a collection of suggestions rather than guidelines, or a set of examples that embody the best practices." [7, Section 28.1].
- "Even if your shop hasn't created explicit coding standards, reviews provide a subtle way of moving toward a group coding standard–decisions are made by the group during reviews, and over time group derives its own standards." [7, Section 28.1].

One could summarize that McConnell advocates that having a consistent coding style as being an advantage in many ways but cautions that the standards should be developed by the programmers in the group and not dictated by nontechnical managers.

E.1.4 Lockheed Martin Joint Strike Fighter C++ Coding Standard

The *Joint Strike Fighter Air Vehicle C++ Coding Standards* document [5] from Lockheed Martin defines C++ coding standards for high consequence applications (i.e. the multi-billion dollar JSF program). While this standard is not the most strict standard out there, it does mandate many different aspects of code formatting such as the placement and indentation of braces '{}' (AV Rules 59, 60, and 61) and the formatting of function prototypes (AV Rule 58). The point is that standards for high consequence (i.e. low

tolerances for defects) may legitimately or otherwise require greater uniformity in source code. While some of the formatting mandates of this document are different than those suggested in [7, Chapter 31], this JSF standard in general is advocated by such individuals as Bjarne Stroustrup [11] and is therefore not without some merit.

E.2 The keyboard analogy for coding styles

The issues involved in going back and forth between different unfamiliar code formatting styles are similar to the issues in going back and forth between different computer keyboard layouts. While some people may naturally prefer one type of keyboard to another (e.g. such as preferring an ergonomic keyboard to avoid problems with repetitive stress injuries or people with larger hands having trouble with smaller keyboards[12]), a person is most proficient when using a single type of keyboard for a long period of time. While a person can generally get used to using a few different types of keyboards that are used frequently (such as the ergonomic keyboard for a desktop computer and a smaller laptop keyboard), having to work occasionally on a very different keyboard really slows down a good typer and increases typing mistakes. For example, a person who uses PC-style keyboards with the Control key on the lower left, are completely handicapped when using a Sun keyboard where the Control key is where the Caps Lock key is on a PC keyboard.

When given enough time, almost anyone can become accustomed to any reasonable keyboard layout and can be productive (as long a unusual physical constraints are not involved). As long as the person uses the keyboard consistently, the productivity will be about the same as with a more favored keyboard layout. Therefore, except for certain physical constraints, a person can learn how to use most keyboard layouts given enough time, but switching back and forth occasionally between different keyboards really damages productivity and increases mistakes.

The same is true for having to read and modify code that uses different code formatting styles. Just about anyone can become accustomed to just about any reasonable coding style if given enough time working with a particular style. However, switching back and forth frequently between different coding styles really does damages productivity and increases coding mistakes for some people, just as switching back and forth between different keyboards can really damage productivity and increase typing mistakes.

E.3 Conclusions

The antagonism between pushing a common formatting style and allowing for individual freedom is similar to a system-wide optimization problem that involves a number of subsystems. In our case, the subsystems are individual coders and the whole system is the team as a whole. Optimizing each subsystem separately would mean that each developer would own and code a district part of the overall system. While this approach maximizes individual developer productivity, it does not maximize overall productivity in that it discourages and hinders collective code ownership that has been demonstrated to be highly effective in the right settings (e.g. Extreme Programming). On the other hand, an overly ridged code formatting standard will allow for collective code ownership but it will also damage the individual productivity of

[11] `http://www.research.att.com/~bs/C++.html`

[12] Computer mice layouts show even greater variability than keyboards and going between different types can hurt productivity even greater. For example, a standard mouse could not be more different than a trackball-type of mouse and going from a standard mouse to a trackball only occasionally can severely degrade productivity if the individual is unfamiliar with the trackball.

every member of the team. Therefore, the “optimal” solution to the code formatting problem is to have the group adopt enough of a uniform style to foster collective code ownership and speed code reviews, but not to needlessly damage individual coder productivity. The balance between these conflicting goals must be handled with care and only group communication along with experience and experimentation will yield a near-optimal solution to the code formatting standards problem for a particular team of developers.

While the above varied sources have different levels of opinions on the importance on consistent code formatting, they all agree that it is the developers themselves that should come up with the guidelines, and not non-technical managers. They also all seem to agree that a coding standard that is too ridged will do more harm than good (i.e. by damaging the productivity and moral of individual programmers).

The majority opinion of these experts, therefore, seems to be that a team of software developers should get together and collectively decide on a sufficient set of guidelines for code formatting and each member should try to follow the spirit of the agreed upon style as much as is reasonable while being allowed to bend or break the guidelines when appropriate.

F Guidelines for reformatting of source code

When a sufficiently common coding style is not being used by all developers in a project and no recommendations for a common coding style exists, then some guidelines are needed for the situations where code written by one individual is modified by another individual that uses a different coding style. These guidelines address how developers should conduct themselves when modifying source files written largely by someone else.

1. First and foremost, each developer should respect the other developers' formatting styles when modifying their code. If a developer has a preferred Emacs style, then that style should be listed explicitly at the top of each source file that is modified. This will help other developers that use Emacs to stay consistent with the file's style.

2. When only small changes are needed, a developer should abide by the formatting style already in use in the file. This helps to respect other developers and helps to avoid needless changes for the version control system to have to track. Again, when user-defined file-specific Emacs styles are specified, then it is easy to maintain a file's style when editing files through Emacs.

3. Reformatting a file written by someone else and checking it in is only justified if significant changes are made. However, if a developer needs to understand a complicated piece of code in order to make perhaps even a small change in the end, then that developer may also be justified in reformatting the file. When a reformatting is done, the new Emacs formatting style should be added to the top of the source file in order to make it easier for the original owner of the file and other developers to maintain the new style.

4. Multiple re-formats of the same file should not be checked in over and over again as this will result in massive increases the the amount of information that the version control system needs to keep track of and makes diffs more difficult to perform.

The above guidelines ensure that individuals are given maximal freedom to format code to their liking but also helps to foster the shared ownership and development of code. In addition, the use of user-defined file-specific formats makes it easy for developers to accommodate formatting styles different from their own.

Sandia National Laboratories